IBM DB2 10.1
for Linux, UNIX, and Windows

Call Level Interface Guide and Reference Volume 2

SC27-3867-00

IBM DB2 10.1
for Linux, UNIX, and Windows

Call Level Interface Guide and Reference Volume 2

SC27-3867-00

Note

Before using this information and the product it supports, read the general information under Appendix B, "Notices," on page 549.

Edition Notice

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at http://www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at http://www.ibm.com/planetwide/

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About this book

The *Call Level Interface (CLI) Guide and Reference* is in two volumes:

- Volume 1 describes how to use CLI to create database applications for DB2® Database for Linux, UNIX, and Windows.
- Volume 2 is a reference that describes CLI functions, keywords and configuration.

Chapter 1. CLI and ODBC function summary

Depr in the ODBC column indicates that the function has been deprecated for ODBC.

The SQL/CLI column can have the following values:

95 The function is defined in the SQL/CLI 9075-3 specification.

SQL3 The function is defined in the SQL/CLI part of the ISO SQL3 draft replacement for SQL/CLI 9075-3.

Table 1. CLI Function list by category

Task Function name	ODBC 3.0	SQL/ CLI	DB2 CLI first version supported	Purpose
Connecting to a data source				
`SQLConnect()`	Depr	95	V1.1	Obtains a connection handle.
`SQLAllocEnv()`	Depr	95	V1.1	Obtains an environment handle. One environment handle is used for one or more connections.
`SQLAllocHandle()`	Core	95	V5	Obtains a handle.
`SQLBrowseConnect()`	Level 1	95	V5	Gets required attributes to connect to a data source.
`SQLConnect()`	Core	95	V1.1	Connects to a specific driver by using a data source name, user ID, and password.
`SQLDriverConnect()`	Core	SQL3	V2.1 [1]	Connects to a specific driver by using a connection string or optionally requests that the Driver Manager and driver display connection dialogs for the user. **Note:** This function is also affected by the additional IBM keywords supported in the `ODBC.INI` file.
`SQLDrivers()`	Core	No	None	CLI does not support this function because this function is implemented by a Driver Manager.
`SQLSetConnectAttr()`	Core	95	V5	Sets connection attributes.
`SQLSetConnectOption()`	Depr	95	V2.1	Sets connection attributes.
`SQLSetConnection()`	No	SQL3	V2.1	Sets the current active connection. You have to use this function only when using embedded SQL within a CLI application with multiple concurrent connections.
Obtaining information about a driver and data source				
SQLDataSources()	Lvl 2	95	V1.1	Returns the list of available data sources.
`SQLGetInfo()`	Core	95	V1.1	Returns information about a specific driver and data source.
`SQLGetFunctions()`	Core	95	V1.1	Returns a list of supported driver functions.

Table 1. CLI Function list by category (continued)

Task Function name	ODBC 3.0	SQL/ CLI	DB2 CLI first version supported	Purpose
SQLGetTypeInfo()	Core	95	V1.1	Returns information about supported data types.
Setting and retrieving driver options				
SQLCreatePkg()	No	No	V9.5	Binds packages to the database.
SQLSetEnvAttr()	Core	95	V2.1	Sets an environment option.
SQLGetEnvAttr()	Core	95	V2.1	Returns the value of an environment option.
SQLGetConnectAttr()	Lvl 1	95	V5	Returns the value of a connection option.
SQLGetConnectOption()	Depr	95	V2.1 [1]	Returns the value of a connection option.
SQLSetStmtAttr()	Core	95	V5	Sets a statement attribute.
SQLSetStmtOption()	Depr	95	V2.1 [1]	Sets a statement option.
SQLGetStmtAttr()	Core	95	V5	Returns the value of a statement attribute.
SQLGetStmtOption()	Depr	95	V2.1 [1]	Returns the value of a statement option.
SQLReloadConfig()	No	No	V9.7	Reloads a configuration property from the client configuration file db2dsdriver.cfg
Preparing SQL requests				
SQLAllocStmt()	Depr	95	V1.1	Allocates a statement handle.
SQLPrepare()	Core	95	V1.1	Prepares an SQL statement for later execution.
SQLExtendedPrepare()	No	No	V6	Prepares an array of statement attributes for an SQL statement for later execution.
SQLExtendedBind()	No	No	V6	Bind an array of columns instead of using repeated calls to SQLBindCol() and SQLBindParameter()
SQLBindParameter()	Lvl 1	95 [2]	V2.1	Assigns storage for a parameter in an SQL statement (ODBC 2.0).
SQLSetParam()	Depr	No	V1.1	Assigns storage for a parameter in an SQL statement (ODBC 1.0). **Note:** In ODBC 2.0, this function has been replaced by SQLBindParameter().
SQLParamOptions()	Depr	No	V2.1	Specifies the use of multiple values for parameters.
SQLGetCursorName()	Core	95	V1.1	Returns the cursor name associated with a statement handle.
SQLSetCursorName()	Core	95	V1.1	Specifies a cursor name.
Submitting requests				
SQLDescribeParam()	Level 2	SQL3	V5	Returns the description of a parameter marker.
SQLExecute()	Core	95	V1.1	Executes a prepared statement.
SQLExecDirect()	Core	95	V1.1	Executes a statement.
SQLNativeSql()	Lvl 2	95	V2.1 [1]	Returns the text of an SQL statement as translated by the driver.

Table 1. CLI Function list by category (continued)

Task Function name	ODBC 3.0	SQL/ CLI	DB2 CLI first version supported	Purpose
SQLNumParams()	Lvl 2	95	V2.1 [1]	Returns the number of parameters in a statement.
SQLParamData()	Lvl 1	95	V2.1 [1]	Used in conjunction with SQLPutData() to supply parameter data at execution time. This is useful for long data values.
SQLPutData()	Core	95	V2.1 [1]	Sends part or all of a data value for a parameter. This is useful for long data values.
Retrieving results and information about results				
SQLRowCount()	Core	95	V1.1	Returns the number of rows affected by an insert, update, or delete request.
SQLNumResultCols()	Core	95	V1.1	Returns the number of columns in the result set.
SQLDescribeCol()	Core	95	V1.1	Describes a column in the result set.
SQLColAttribute()	Core	Yes	V5	Describes attributes of a column in the result set.
SQLColAttributes()	Depr	Yes	V1.1	Describes attributes of a column in the result set.
SQLColumnPrivileges()	Level 2	95	V2.1	Gets privileges associated with the columns of a table.
SQLSetColAttributes()	No	No	V2.1	Sets attributes of a column in the result set.
SQLBindCol()	Core	95	V1.1	Assigns storage for a result column and specifies the data type.
SQLFetch()	Core	95	V1.1	Returns a result row.
SQLFetchScroll()	Core	95	V5	Returns a rowset from a result row.
SQLExtendedFetch()	Depr	95	V2.1	Returns multiple result rows.
SQLGetData()	Core	95	V1.1	Returns part or all of one column of one row of a result set. This is useful for long data values.
SQLMoreResults()	Lvl 1	SQL3	V2.1 [a]	Determines whether there are more result sets available and, if so, initializes processing for the next result set.
SQLNextResult()	No	Yes	V7.1	Provides nonsequential access to multiple result sets returned from a stored procedure.
SQLError()	Depr	95	V1.1	Returns additional error or status information.
SQLGetDiagField()	Core	95	V5	Gets a field of diagnostic data.
SQLGetDiagRec()	Core	95	V5	Gets multiple fields of diagnostic data.
SQLSetPos()	Level 1	SQL3	V5	Sets the cursor position in a rowset.
SQLGetSQLCA()	No	No	V2.1	Returns the SQLCA associated with a statement handle.
SQLBulkOperations()	Level 1	No	V6	Performs bulk insertions, updates, deletions, and fetches by bookmark.

Table 1. CLI Function list by category (continued)

Task Function name	ODBC 3.0	SQL/ CLI	DB2 CLI first version supported	Purpose
Descriptors				
SQLCopyDesc()	Core	95	V5	Copies descriptor information between handles.
SQLGetDescField()	Core	95	V5	Gets single field settings of a descriptor record.
SQLGetDescRec()	Core	95	V5	Gets multiple field settings of a descriptor record.
SQLSetDescField()	Core	95	V5	Sets a single field of a descriptor record.
SQLSetDescRec()	Core	95	V5	Sets multiple field settings of a descriptor record.
Large object support				
SQLBindFileToCol()	No	No	V2.1	Associates a LOB file reference with a LOB column.
SQLBindFileToParam()	No	No	V2.1	Associates a LOB file reference with a parameter marker.
SQLGetLength()	No	SQL3	V2.1	Gets the length of a string referenced by a LOB locator.
SQLGetPosition()	No	SQL3	V2.1	Gets the position of a string within a source string referenced by a LOB locator.
SQLGetSubString()	No	SQL3	V2.1	Creates a new LOB locator that references a substring within a source string. The source string is also represented by a LOB locator.
Obtaining information about the data source's system tables (catalog functions)				
SQLColumns()	Lvl 1	SQL3	V2.1 [1]	Returns the list of column names in specified tables.
SQLExtendedProcedures()	No	No	V9.7	Returns the list of procedure names stored in a specific data source with additional information.
SQLExtendedProceduresColumns()	No	No	V9.7	Returns the list of input and output parameters for the specified procedures with additional information.
SQLForeignKeys()	Lvl 2	SQL3	V2.1	Returns the list of column names that comprise foreign keys, if they exist for a specified table.
SQLPrimaryKeys()	Lvl 1	SQL3	V2.1	Returns the list of column names that comprise the primary key for a table.
SQLProcedureColumns()	Lvl 2	No	V2.1	Returns the list of input and output parameters for the specified procedures.
SQLProcedures()	Lvl 2	No	V2.1	Returns the list of procedure names stored in a specific data source.
SQLSpecialColumns()	Core	SQL3	V2.1 [1]	Returns information about the optimal set of columns that uniquely identifies a row in a specified table.

Table 1. CLI Function list by category (continued)

Task Function name	ODBC 3.0	SQL/ CLI	DB2 CLI first version supported	Purpose
`SQLStatistics()`	Core	SQL3	V2.1 [1]	Returns statistics about a single table and the list of indexes associated with the table.
`SQLTablePrivileges()`	Lvl 2	SQL3	V2.1	Returns a list of tables and the privileges associated with each table.
`SQLTables()`	Core	SQL3	V2.1 [1]	Returns the list of table names stored in a specific data source.
Terminating a statement				
`SQLFreeHandle()`	Core	95	V1.1	Frees handle resources.
`SQLFreeStmt()`	Core	95	V1.1	Ends statement processing and closes the associated cursor, discards pending results, and, optionally, frees all resources associated with the statement handle.
`SQLCancel()`	Core	95	V1.1	Cancels an SQL statement.
`SQLTransact()`	Depr	No	V1.1	Commits or rolls back a transaction.
`SQLCloseCursor()`	Core	95	V5	Commits or rolls back a transaction.
Terminating a connection				
`SQLDisconnect()`	Core	95	V1.1	Closes the connection.
`SQLEndTran()`	Core	95	V5	Ends the transaction of a connection.
`SQLFreeConnect()`	Depr	95	V1.1	Releases the connection handle.
`SQLFreeEnv()`	Depr	95	V1.1	Releases the environment handle.
Creating and dropping a database				
`SQLCreateDb()`	No	No	V9.7	Creates a database based on the specified database name, code-set, and mode.
`SQLDropDb()`	No	No	V9.7	Drops the specified database.

Note:

[1] Runtime support for this function was also available in the DB2 Client Application Enabler for DOS Version 1.2 product.

[2] `SQLBindParam()` has been replaced by `SQLBindParameter()`.

The following limitations apply to ODBC functions:

- SQLSetScrollOptions() is supported for runtime use only, because it has been superseded by the SQL_CURSOR_TYPE, SQL_CONCURRENCY, SQL_KEYSET_SIZE, and SQL_ROWSET_SIZE statement options.
- `SQLDrivers()` is implemented by the ODBC driver manager.

Unicode functions (CLI)

CLI Unicode functions accept Unicode string arguments in place of ANSI string arguments. The Unicode string arguments must be in UCS-2 encoding (native-endian format). ODBC API functions have suffixes to indicate the format of their string arguments: those that accept Unicode end in W, and those that accept ANSI have no suffix (ODBC adds equivalent functions with names that end in A, but these are not offered by CLI). The following list of CLI functions are available in both ANSI and Unicode versions:

- SQLBrowseConnect
- SQLColAttribute
- SQLColAttributes
- SQLColumnPrivileges
- SQLColumns
- SQLConnect
- SQLCreateDb
- SQLDataSources
- SQLDescribeCol
- SQLDriverConnect
- SQLDropDb
- SQLError
- SQLExecDirect
- SQLExtendedPrepare
- SQLExtendedProcedures
- SQLExtendedProcedureColumns
- SQLForeignKeys
- SQLGetConnectAttr
- SQLGetConnectOption
- SQLGetCursorName
- SQLGetDescField
- SQLGetDescRec
- SQLGetDiagField
- SQLGetDiagRec
- SQLGetInfo
- SQLGetPosition
- SQLGetStmtAttr
- SQLNativeSQL
- SQLPrepare
- SQLPrimaryKeys
- SQLProcedureColumns
- SQLProcedures
- SQLReloadConfig
- SQLSetConnectAttr
- SQLSetConnectOption
- SQLSetCursorName
- SQLSetDescField
- SQLSetStmtAttr
- SQLSpecialColumns
- SQLStatistics
- SQLTablePrivileges
- SQLTables

Unicode functions that have arguments which are always the length of strings interpret these arguments as the number of SQLWCHAR elements needed to store the string. For functions that return length information for server data, the display

size and precision are again described in terms of the number of SQLWCHAR elements used to store them. When the length (transfer size of the data) can refer to string or non-string data, it is interpreted as the number of bytes needed to store the data.

For example, `SQLGetInfoW()` will still take the length as the number of bytes, but `SQLExecDirectW()` will use the number of SQLWCHAR elements. Consider a single character from the UTF-16 extended character set (UTF-16 is an extended character set of UCS-2; Microsoft Windows 2000 and Microsoft Windows XP use UTF-16). Microsoft Windows 2000 will use two SQL_C_WCHAR elements, which is equivalent to 4 bytes, to store this single character. The character therefore has a display size of 1, a string length of 2 (when using SQL_C_WCHAR), and a byte count of 4. CLI will return data from result sets in either Unicode or ANSI, depending on the application's binding. If an application binds to SQL_C_CHAR, the driver will convert SQL_WCHAR data to SQL_CHAR. An ODBC driver manager, if used, maps SQL_C_WCHAR to SQL_C_CHAR for ANSI drivers but does no mapping for Unicode drivers.

ANSI to Unicode function mappings

The syntax for a CLI Unicode function is the same as the syntax for its corresponding ANSI function, except that SQLCHAR parameters are defined as SQLWCHAR. Character buffers defined as SQLPOINTER in the ANSI syntax can be defined as either SQLCHAR or SQLWCHAR in the Unicode function. Refer to the ANSI version of the CLI Unicode functions for ANSI syntax details.

SQLAllocConnect function (CLI) - Allocate connection handle

In ODBC 3.0, SQLAllocConnect() has been deprecated and replaced with SQLAllocHandle().

Although this version of CLI continues to support SQLAllocConnect(), it is recommended that you use SQLAllocHandle() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

The statement:

```
SQLAllocConnect(henv, &hdbc);
```

for example, would be rewritten using the new function as:

```
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
```

SQLAllocEnv function (CLI) - Allocate environment handle

In ODBC 3.0, SQLAllocEnv() has been deprecated and replaced with `SQLAllocHandle()`.

Although this version of CLI continues to support SQLAllocEnv(), use `SQLAllocHandle()` in your CLI programs so that they conform to the latest standards.

Migrating to the new function

The statement:

```
SQLAllocEnv(&henv);
```

for example, would be rewritten using the new function as:

```
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
```

SQLAllocHandle function (CLI) - Allocate handle

Allocates environment, connection, statement, or descriptor handles.

Note: This function replaces the deprecated ODBC 2.0 functions SQLAllocConnect(), SQLAllocEnv(), and SQLAllocStmt().

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Syntax

```
SQLRETURN   SQLAllocHandle (
               SQLSMALLINT       HandleType,         /* fHandleType */
               SQLHANDLE         InputHandle,        /* hInput */
               SQLHANDLE         *OutputHandlePtr);  /* *phOutput */
```

Function Arguments

Table 2. SQLAllocHandle arguments

Data type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	The type of handle to be allocated by `SQLAllocHandle()`. Must be one of the following values: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC
SQLHANDLE	*InputHandle*	input	Existing handle to use as a context for the new handle being allocated. If HandleType is SQL_HANDLE_ENV, this is SQL_NULL_HANDLE. If HandleType is SQL_HANDLE_DBC, this must be an environment handle, and if it is SQL_HANDLE_STMT or SQL_HANDLE_DESC, it must be a connection handle.
SQLHANDLE *	*OutputHandlePtr*	output	Pointer to a buffer in which to return the handle to the newly allocated data structure.

Usage

SQLAllocHandle() is used to allocate environment, connection, statement, and descriptor handles. An application can allocate multiple environment, connection, statement, or descriptor handles at any time a valid *InputHandle* exists.

If the application calls SQLAllocHandle() with **OutputHandlePtr* set to an existing environment, connection, statement, or descriptor handle, CLI overwrites the handle, and new resources appropriate to the handle type are allocated. There are no changes made to the CLI resources associated with the original handle.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

If SQLAllocHandle() returns SQL_INVALID_HANDLE, it will set *OutputHandlePtr* to SQL_NULL_HENV, SQL_NULL_HDBC, SQL_NULL_HSTMT, or SQL_NULL_HDESC, depending on the value of *HandleType*, unless the output argument is a null pointer. The application can then obtain additional information from the diagnostic data structure associated with the handle in the *InputHandle* argument.

Diagnostics

Table 3. SQLAllocHandle SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	The HandleType argument was SQL_HANDLE_STMT or SQL_HANDLE_DESC, but the connection handle specified by the *InputHandle* argument did not have an open connection. The connection process must be completed successfully (and the connection must be open) for CLI to allocate a statement or descriptor handle.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY013	Unexpected memory handling error.	The *HandleType* argument was SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC; and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY014	No more handles.	The limit for the number of handles that can be allocated for the type of handle indicated by the *HandleType* argument has been reached, or in some cases, insufficient system resources exist to properly initialize the new handle.
HY092	Option type out of range.	The *HandleType* argument was not one of: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC

Restrictions

None.

Example

```
SQLHANDLE henv; /* environment handle */
SQLHANDLE hdbc; /* connection handle */
SQLHANDLE hstmt; /* statement handle */
SQLHANDLE hdesc; /* descriptor handle */

/* ... */

/* allocate an environment handle */
cliRC = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

/* ... */

/* allocate a database connection handle */
cliRC = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

/* ... */
/* connect to database using hdbc */
/* ... */

/* allocate one or more statement handles */
cliRC = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

/* ... */
/* allocate a descriptor handle */
cliRC = SQLAllocHandle(SQL_HANDLE_DESC, hstmt, &hdesc);
```

SQLAllocStmt function (CLI) - Allocate a statement handle

In ODBC 3.0, SQLAllocStmt() has been deprecated and replaced with SQLAllocHandle().

Although this version of CLI continues to support SQLAllocStmt(), use SQLAllocHandle() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

The statement:

```
SQLAllocStmt(hdbc, &hstmt);
```

for example, would be rewritten using the new function as:

```
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
```

SQLBindCol function (CLI) - Bind a column to an application variable or LOB locator

Application can associates (bind) columns in a result set to C data type variables, and associate (bind) LOB columns in a result set to LOB locators.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

SQLBindCol function (CLI) - Bind a column to an application variable or LOB locator

SQLBindCol() is used to associate columns in a result set to either:

- Application variables or arrays of application variables (storage buffers), for all C data types. Data is transferred from the DBMS to the application when SQLFetch() or SQLFetchScroll() is called. Data conversion might occur as the data is transferred.
- A LOB locator, for LOB columns. A LOB locator, not the data itself, is transferred from the DBMS to the application when SQLFetch() is called.

  Alternatively, LOB columns can be bound directly to a file using SQLBindFileToCol().

SQLBindCol() is called once for each column in the result set that the application needs to retrieve.

In general, SQLPrepare(), SQLExecDirect() or one of the schema functions is called before this function, and SQLFetch(), SQLFetchScroll(), `SQLBulkOperations()`, or `SQLSetPos()` is called after. Column attributes might also be needed before calling SQLBindCol(), and can be obtained using SQLDescribeCol() or SQLColAttribute().

Syntax

```
SQLRETURN   SQLBindCol (
               SQLHSTMT          StatementHandle,          /* hstmt */
               SQLUSMALLINT      ColumnNumber,             /* icol */
               SQLSMALLINT       TargetType,               /* fCType */
               SQLPOINTER        TargetValuePtr,           /* rgbValue */
               SQLLEN            BufferLength,             /* dbValueMax */
               SQLLEN            *StrLen_or_IndPtr);       /* *pcbValue */
```

Function arguments

Table 4. SQLBindCol arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLUSMALLINT	*ColumnNumber*	input	Number identifying the column. Columns are numbered sequentially, from left to right. • Column numbers start at 1 if bookmarks are not used (SQL_ATTR_USE_BOOKMARKS statement attribute set to SQL_UB_OFF). • Column numbers start at 0 if bookmarks are used (the statement attribute is set to SQL_UB_ON). Column 0 is the bookmark column.

Table 4. SQLBindCol arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*TargetType*	input	The C data type for column number *ColumnNumber* in the result set. When the application retrieves data from the data source, it will convert the data to this C type. When using `SQLBulkOperations()` or `SQLSetPos()`, the driver will convert data from this C data type when sending information to the data source. The following types are supported: • SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CHAR • SQL_C_CLOB_LOCATOR • SQL_C_DBCHAR • SQL_C_DBCLOB_LOCATOR • SQL_C_DECIMAL_IBM • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_NUMERIC [a] • SQL_C_SBIGINT • SQL_C_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TYPE_TIMESTAMP_EXT • SQL_C_TINYINT • SQL_C_UBIGINT • SQL_C_UTINYINT • SQL_C_WCHAR Specifying SQL_C_DEFAULT causes data to be transferred to its default C data type.
SQLPOINTER	*TargetValuePtr*	input/output (deferred)	Pointer to buffer or an array of buffers with either column-wise or row-wise binding, where CLI is to store the column data or the LOB locator when the fetch occurs. This buffer is used to return data when any of the following functions are called: `SQLFetch()`, `SQLFetchScroll()`, `SQLSetPos()` using the *Operation* argument SQL_REFRESH, or `SQLBulkOperations()` using the *Operation* argument SQL_FETCH_BY_BOOKMARK. Otherwise, `SQLBulkOperations()` and `SQLSetPos()` use the buffer to retrieve data. If *TargetValuePtr* is null, the column is unbound. All columns can be unbound with a call to `SQLFreeStmt()` with the SQL_UNBIND option.

Table 4. SQLBindCol arguments (continued)

Data type	Argument	Use	Description
SQLLEN	*BufferLength*	input	Size in bytes of *TargetValuePtr* buffer available to store the column data or the LOB locator. If *TargetType* denotes a binary or character string (either single or double byte) or is SQL_C_DEFAULT for a column returning variable length data, then *BufferLength* must be > 0, or an error will be returned. Note that for character data, the driver counts the NULL termination character and so space must be allocated for it. For all other data types, this argument is ignored.
SQLLEN *	*StrLen_or_IndPtr*	input/output (deferred)	Pointer to value (or array of values) which indicates the number of bytes CLI has available to return in the *TargetValuePtr* buffer. If *TargetType* is a LOB locator, the size of the locator is returned, not the size of the LOB data. This buffer is used to return data when any of the following functions are called: `SQLFetch()`, `SQLFetchScroll()`, `SQLSetPos()` using the *Operation* argument SQL_REFRESH, or `SQLBulkOperations()` using the *Operation* argument SQL_FETCH_BY_BOOKMARK. Otherwise, `SQLBulkOperations()` and `SQLSetPos()` use the buffer to retrieve data. SQLFetch() returns SQL_NULL_DATA in this argument if the data value of the column is null. This pointer value must be unique for each bound column, or NULL. A value of SQL_COLUMN_IGNORE, SQL_NTS, SQL_NULL_DATA, or the length of the data can be set for use with `SQLBulkOperations()`. SQL_NO_LENGTH might also be returned, refer to the Usage section for more information.

- For this function, both *TargetValuePtr* and *StrLen_or_IndPtr* are deferred outputs, meaning that the storage locations these pointers point to do not get updated until a result set row is fetched. As a result, the locations referenced by these pointers must remain valid until SQLFetch() or SQLFetchScroll() is called. For example, if SQLBindCol() is called within a local function, SQLFetch() must be called from within the same scope of the function or the *TargetValuePtr* buffer must be allocated as static or global.
- CLI will be able to optimize data retrieval for all variable length data types if *TargetValuePtr* is placed consecutively in memory after *StrLen_or_IndPtr*.

Usage

Call SQLBindCol() once for each column in the result set for which either the data or, for LOB columns, the LOB locator is to be retrieved. When SQLFetch() or SQLFetchScroll() is called to retrieve data from the result set, the data in each of the bound columns is placed in the locations assigned by the *TargetValuePtr* and *StrLen_or_IndPtr* pointers. When the statement attribute SQL_ATTR_ROW_ARRAY_SIZE is greater than 1, then *TargetType* should refer to

an array of buffers. If *TargetType* is a LOB locator, a locator value is returned, not the actual LOB data. The LOB locator references the entire data value in the LOB column.

If a CLI application does not provide an output buffer for a LOB column using the function SQLBindCol() the IBM® data server client will, by default, request a LOB locator on behalf of the application for each LOB column in the result sets.

Columns are identified by a number, assigned sequentially from left to right.
- Column numbers start at 1 if bookmarks are not used (SQL_ATTR_USE_BOOKMARKS statement attribute set to SQL_UB_OFF).
- Column numbers start at 0 if bookmarks are used (the statement attribute set to SQL_UB_ON).

After columns have been bound, in subsequent fetches the application can change the binding of these columns or bind previously unbound columns by calling SQLBindCol(). The new binding does not apply to data already fetched, it will be used on the next fetch. To unbind a single column (including columns bound with SQLBindFileToCol()), call SQLBindCol() with the *TargetValuePtr* pointer set to NULL. To unbind all the columns, the application should call SQLFreeStmt() with the *Option* input set to SQL_UNBIND.

The application must ensure enough storage is allocated for the data to be retrieved. If the buffer is to contain variable length data, the application must allocate as much storage as the maximum length of the bound column plus the NULL terminator. Otherwise, the data might be truncated. If the buffer is to contain fixed length data, CLI assumes the size of the buffer is the length of the C data type. If data conversion is specified, the required size might be affected.

If string truncation does occur, SQL_SUCCESS_WITH_INFO is returned and *StrLen_or_IndPtr* will be set to the actual size of *TargetValuePtr* available for return to the application.

Truncation is also affected by the SQL_ATTR_MAX_LENGTH statement attribute (used to limit the amount of data returned to the application). The application can specify not to report truncation by calling SQLSetStmtAttr() with SQL_ATTR_MAX_LENGTH and a value for the maximum length to return for all variable length columns, and by allocating a *TargetValuePtr* buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, SQL_SUCCESS will be returned when the value is fetched and the maximum length, not the actual length, will be returned in *StrLen_or_IndPtr*.

If the column to be bound is a SQL_GRAPHIC, SQL_VARGRAPHIC or SQL_LONGVARGRAPHIC type, then *TargetType* can be set to SQL_C_DBCHAR or SQL_C_CHAR. If *TargetType* is SQL_C_DBCHAR, the data fetched into the *TargetValuePtr* buffer will be null-terminated with a double byte null-terminator. If *TargetType* is SQL_C_CHAR, then there will be no null-termination of the data. In both cases, the length of the *TargetValuePtr* buffer (*BufferLength*) is in units of bytes and should therefore be a multiple of 2. It is also possible to force CLI to null terminate graphic strings using the PATCH1 keyword.

Note: SQL_NO_TOTAL will be returned in *StrLen_or_IndPtr* if:
- The SQL type is a variable length type, and
- *StrLen_or_IndPtr* and *TargetValuePtr* are contiguous, and
- The column type is NOT NULLABLE, and
- String truncation occurred.

Descriptors and SQLBindCol

The following sections describe how SQLBindCol() interacts with descriptors.

Note: Calling SQLBindCol() for one statement can affect other statements. This occurs when the ARD associated with the statement is explicitly allocated and is also associated with other statements. Because SQLBindCol() modifies the descriptor, the modifications apply to all statements with which this descriptor is associated. If this is not the required behavior, the application should dissociate this descriptor from the other statements before calling SQLBindCol().

Argument mappings

Conceptually, SQLBindCol() performs the following steps in sequence:
- Calls SQLGetStmtAttr() to obtain the ARD handle.
- Calls SQLGetDescField() to get this descriptor's SQL_DESC_COUNT field, and if the value in the *ColumnNumber* argument exceeds the value of SQL_DESC_COUNT, calls SQLSetDescField() to increase the value of SQL_DESC_COUNT to *ColumnNumber*.
- Calls SQLSetDescField() multiple times to assign values to the following fields of the ARD:
  - Sets SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE to the value of *TargetType*.
  - Sets one or more of SQL_DESC_LENGTH, SQL_DESC_PRECISION, SQL_DESC_SCALE as appropriate for *TargetType*.
  - Sets the SQL_DESC_OCTET_LENGTH field to the value of *BufferLength*.
  - Sets the SQL_DESC_DATA_PTR field to the value of *TargetValue*.
  - Sets the SQL_DESC_INDICATOR_PTR field to the value of *StrLen_or_IndPtr* (see the following paragraph).
  - Sets the SQL_DESC_OCTET_LENGTH_PTR field to the value of *StrLen_or_IndPtr* (see the following paragraph).

The variable that the *StrLen_or_IndPtr* argument refers to is used for both indicator and length information. If a fetch encounters a null value for the column, it stores SQL_NULL_DATA in this variable; otherwise, it stores the data length in this variable. Passing a null pointer as *StrLen_or_IndPtr* keeps the fetch operation from returning the data length, but makes the fetch fail if it encounters a null value and has no way to return SQL_NULL_DATA.

If the call to SQLBindCol() fails, the content of the descriptor fields it would have set in the ARD are undefined, and the value of the SQL_DESC_COUNT field of the ARD is unchanged.

Implicit resetting of COUNT field

SQLBindCol() sets SQL_DESC_COUNT to the value of the *ColumnNumber* argument only when this would increase the value of SQL_DESC_COUNT. If the value in the *TargetValuePtr* argument is a null pointer and the value in the *ColumnNumber* argument is equal to SQL_DESC_COUNT (that is, when unbinding the highest bound column), then SQL_DESC_COUNT is set to the number of the highest remaining bound column.

Cautions regarding SQL_C_DEFAULT

To retrieve column data successfully, the application must determine correctly the length and starting point of the data in the application buffer. When the

application specifies an explicit *TargetType*, application misconceptions are readily detected. However, when the application specifies a *TargetType* of SQL_C_DEFAULT, SQLBindCol() can be applied to a column of a different data type from the one intended by the application, either from changes to the metadata or by applying the code to a different column. In this case, the application might fail to determine the start or length of the fetched column data. This can lead to unreported data errors or memory violations.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 5. SQLBindCol SQLSTATEs

SQLSTATE	Description	Explanation
07009	Invalid descriptor index	The value specified for the argument *ColumnNumber* exceeded the maximum number of columns in the result set, or the value specified was less than 0.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY003	Program type out of range.	*TargetType* was not a valid data type or SQL_C_DEFAULT.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* is less than 1 and the argument *TargetType* is either SQL_C_CHAR, SQL_C_BINARY or SQL_C_DEFAULT.
HYC00	Driver not capable.	CLI recognizes, but does not support the data type specified in the argument *TargetType* A LOB locator C data type was specified, but the connected server does not support LOB data types.

Note: Additional diagnostic messages relating to the bound columns might be reported at fetch time.

Restrictions

The LOB data support is only available when connected to a server that supports large object data types. If the application attempts to specify a LOB locator C data type for a server that does not support it, SQLSTATE **HY**C00 will be returned.

Example

```
/* bind column 1 to variable */
cliRC = SQLBindCol(hstmt, 1, SQL_C_SHORT, &deptnumb.val, 0, &deptnumb.ind);
```

SQLBindFileToCol function (CLI) - Bind LOB file reference to LOB column

Associates or binds a LOB or XML column in a result set to a file reference or an array of file references.

This enables data in that column to be transferred directly into a file when each row is fetched for the statement handle.

Specification:

- **CLI** 2.1

The LOB file reference arguments (file name, file name length, file reference options) refer to a file within the application's environment (on the client). Before fetching each row, the application must make sure that these variables contain the name of a file, the length of the file name, and a file option (new / overwrite / append). These values can be changed between each row fetch operation.

Syntax

```
SQLRETURN   SQLBindFileToCol (SQLHSTMT          StatementHandle,   /* hstmt */
                              SQLUSMALLINT      ColumnNumber,      /* icol */
                              SQLCHAR           *FileName,
                              SQLSMALLINT       *FileNameLength,
                              SQLUINTEGER       *FileOptions,
                              SQLSMALLINT       MaxFileNameLength,
                              SQLINTEGER        *StringLength,
                              SQLINTEGER        *IndicatorValue);
```

Function arguments

Table 6. SQLBindFileToCol arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLUSMALLINT	*icol*	input	Number identifying the column. Columns are numbered sequentially, from left to right, starting at 1.
SQLCHAR *	*FileName*	input (deferred)	Pointer to the location that will contain the file name or an array of file names at the time of the next fetch using the *StatementHandle*. This is either the complete path name of the file(s) or a relative file name(s). If relative file name(s) are provided, they are appended to the current path of the running application. This pointer cannot be NULL.
SQLSMALLINT *	*FileNameLength*	input (deferred)	Pointer to the location that will contain the length of the file name (or an array of lengths) at the time of the next fetch using the *StatementHandle*. If this pointer is NULL, then the *FileName* will be considered a null-terminated string, similar to passing a length of SQL_NTS. The maximum value of the file name length is 255.

Table 6. SQLBindFileToCol arguments (continued)

Data type	Argument	Use	Description
SQLUINTEGER *	*FileOptions*	input (deferred)	Pointer to the location that will contain the file option or (array of file options) to be used when writing the file at the time of the next fetch using the *StatementHandle*. The following *FileOptions* are supported: **SQL_FILE_CREATE** Create a new file. If a file by this name already exists, SQL_ERROR will be returned. **SQL_FILE_OVERWRITE** If the file already exists, overwrite it. Otherwise, create a new file. **SQL_FILE_APPEND** If the file already exists, append the data to it. Otherwise, create a new file. Only one option can be chosen per file, there is no default.
SQLSMALLINT	*MaxFileNameLength*	input	This specifies the length of the *FileName* buffer or, if the application uses SQLFetchScroll() to retrieve multiple rows for the LOB column, this specifies the length of each element in the *FileName* array.
SQLINTEGER *	*StringLength*	output (deferred)	Pointer to the location that contains the length (or array of lengths) in bytes of the LOB data that is returned. If this pointer is NULL, nothing is returned.
SQLINTEGER *	*IndicatorValue*	output (deferred)	Pointer to the location that contains an indicator value (or array of values).

Usage

The application calls SQLBindFileToCol() once for each column that should be transferred directly to a file when a row is fetched. LOB data is written directly to the file without any data conversion, and without appending null-terminators. XML data is written out in UTF-8, with an XML declaration generated according to the setting of the SQL_ATTR_XML_DECLARATION connection or statement attribute.

FileName, *FileNameLength*, and *FileOptions* must be set before each fetch. When SQLFetch() or SQLFetchScroll() is called, the data for any column which has been bound to a LOB file reference is written to the file or files pointed to by that file reference. Errors associated with the deferred input argument values of SQLBindFileToCol() are reported at fetch time. The LOB file reference, and the deferred *StringLength* and *IndicatorValue* output arguments are updated between fetch operations.

If SQLFetchScroll() is used to retrieve multiple rows for the LOB column, *FileName*, *FileNameLength*, and *FileOptions* point to arrays of LOB file reference variables. In this case, *MaxFileNameLength* specifies the length of each element in the *FileName* array and is used by CLI to determine the location of each element in the *FileName* array. The contents of the array of file references must be valid at the time of the

SQLFetchScroll() call. The *StringLength* and *IndicatorValue* pointers each point to an array whose elements are updated upon the SQLFetchScroll() call.

Using SQLFetchScroll(), multiple LOB values can be written to multiple files, or to the same file depending on the file names specified. If writing to the same file, the SQL_FILE_APPEND file option should be specified for each file name entry. Only column-wise binding of arrays of file references is supported with SQLFetchScroll().

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 7. SQLBindFileToCol SQLSTATEs

SQLSTATE	Description	Explanation
07009	Invalid column number.	The value specified for the argument *icol* was less than 1. The value specified for the argument *icol* exceeded the maximum number of columns supported by the data source.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY009	Invalid argument value.	*FileName*, *StringLength* or *FileOptions* is a null pointer.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument *MaxFileNameLength* was less than 0.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.

Restrictions

This function is not available when connected to DB2 servers that do not support large object data types. Call SQLGetFunctions() with the function type set to SQL_API_SQLBINDFILETOCOL and check the *SupportedPtr* output argument to determine if the function is supported for the current connection.

Example

```
/* bind a file to the BLOB column */
  rc = SQLBindFileToCol(hstmt,
                        1,
                        fileName,
                        &fileNameLength,
                        &fileOption,
                        14,
                        NULL,
                        &fileInd);
```

SQLBindFileToParam function (CLI) - Bind LOB file reference to LOB parameter

SQLBindFileToParam() is used to associate or bind a parameter marker in an SQL statement to a file reference or an array of file references. This enables data from the file to be transferred directly into a LOB or XML column when the statement is subsequently executed.

Specification:

- **CLI** 2.1

The LOB file reference arguments (file name, file name length, file reference options) refer to a file within the application's environment (on the client). Before calling SQLExecute() or SQLExecDirect(), the application must make sure that this information is available in the deferred input buffers. These values can be changed between SQLExecute() calls.

Syntax

```
SQLRETURN SQLBindFileToParam (
            SQLHSTMT          StatementHandle,          /* hstmt */
            SQLUSMALLINT      TargetType,               /* ipar */
            SQLSMALLINT       DataType,                 /* fSqlType */
            SQLCHAR           *FileName,
            SQLSMALLINT       *FileNameLength,
            SQLUINTEGER       *FileOptions,
            SQLSMALLINT       MaxFileNameLength,
            SQLINTEGER        *IndicatorValue);
```

Function arguments

Table 8. SQLBindFileToParam arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLUSMALLINT	*TargetType*	input	Parameter marker number. Parameters are numbered sequentially, from left to right, starting at 1.
SQLSMALLINT	*DataType*	input	SQL Data Type of the column. The data type must be one of: • SQL_BLOB • SQL_CLOB • SQL_DBCLOB • SQL_XML

Table 8. SQLBindFileToParam arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*FileName*	input (deferred)	Pointer to the location that will contain the file name or an array of file names when the statement (*StatementHandle*) is executed. This is either the complete path name of the file or a relative file name. If a relative file name is provided, it is appended to the current path of the client process. This argument cannot be NULL.
SQLSMALLINT *	*FileNameLength*	input (deferred)	Pointer to the location that will contain the length of the file name (or an array of lengths) at the time of the next `SQLExecute()` or `SQLExecDirect()` using the *StatementHandle*. If this pointer is NULL, then the *FileName* will be considered a null-terminated string, similar to passing a length of SQL_NTS. The maximum value of the file name length is 255.
SQLUINTEGER *	*FileOptions*	input (deferred)	Pointer to the location that will contain the file option (or an array of file options) to be used when reading the file. The location will be accessed when the statement (*StatementHandle*) is executed. Only one option is supported (and it must be specified): **SQL_FILE_READ** A regular file that can be opened, read and closed. (The length is computed when the file is opened) This pointer cannot be NULL.
SQLSMALLINT	*MaxFileNameLength*	input	This specifies the length of the *FileName* buffer. If the application calls `SQLParamOptions()` to specify multiple values for each parameter, this is the length of each element in the *FileName* array.
SQLINTEGER *	*IndicatorValue*	input (deferred)	Pointer to the location that contains an indicator value (or array of values), which is set to SQL_NULL_DATA if the data value of the parameter is to be null. It must be set to 0 (or the pointer can be set to null) when the data value is not null.

Usage

The application calls SQLBindFileToParam() once for each parameter marker whose value should be obtained directly from a file when a statement is executed. Before the statement is executed, *FileName*, *FileNameLength*, and *FileOptions* values must be set. When the statement is executed, the data for any parameter which has been bound using SQLBindFileToParam() is read from the referenced file and passed to the server.

If the application uses `SQLParamOptions()` to specify multiple values for each parameter, then *FileName, FileNameLength,* and *FileOptions* point to an array of LOB file reference variables. In this case, *MaxFileNameLength* specifies the length of each element in the *FileName* array and is used by CLI to determine the location of each element in the *FileName* array.

A LOB parameter marker can be associated with (bound to) an input file using SQLBindFileToParam(), or with a stored buffer using `SQLBindParameter()`. The most recent bind parameter function call determines the type of binding that is in effect.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 9. SQLBindFileToParam SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY004	SQL data type out of range.	The value specified for *DataType* was not a valid SQL type for this function call.
HY009	Invalid argument value.	*FileName*, *FileOptions FileNameLength*, is a null pointer.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the input argument *MaxFileNameLength* was less than 0.
HY093	Invalid parameter number.	The value specified for *TargetType* was either less than 1 or greater than the maximum number of parameters supported.
HYC00	Driver not capable.	The server does not support Large Object data types.

Restrictions

This function is not available when connected to DB2 servers that do not support large object data types. Call `SQLGetFunctions()` with the function type set to SQL_API_SQLBINDFILETOPARAM and check the *SupportedPtr* output argument to determine if the function is supported for the current connection.

Example

```
/* bind the file parameter */
rc = SQLBindFileToParam(hstmt,
                        3,
                        SQL_BLOB,
                        fileName,
```

```
&fileNameLength,
&fileOption,
14,
&fileInd);
```

SQLBindParameter function (CLI) - Bind a parameter marker to a buffer or LOB locator

Binds parameter markers in an SQL statement to application variables, arrays of application variables, or lob locators.

Specification:

- **CLI** 2.1
- **ODBC** 2.0

SQLBindParameter() binds parameter markers to either:

- Application variables or arrays of application variables (storage buffers) for all C data types. In this case data is transferred from the application to the DBMS when SQLExecute() or SQLExecDirect() is called. Data conversion might occur as the data is transferred.
- A LOB locator, for SQL LOB data types. In this case a LOB locator value, not the LOB data itself, is transferred from the application to the server when the SQL statement is executed.

  Alternatively, LOB parameters can be bound directly to a file using `SQLBindFileToParam()`

This function must also be used to bind a parameter of a stored procedure CALL statement to the application where the parameter can be input, output or both.

Syntax

```
SQLRETURN  SQLBindParameter(
              SQLHSTMT          StatementHandle,     /* hstmt */
              SQLUSMALLINT      ParameterNumber,     /* ipar */
              SQLSMALLINT       InputOutputType,     /* fParamType */
              SQLSMALLINT       ValueType,           /* fCType */
              SQLSMALLINT       ParameterType,       /* fSqlType */
              SQLULEN           ColumnSize,          /* cbColDef */
              SQLSMALLINT       DecimalDigits,       /* ibScale */
              SQLPOINTER        ParameterValuePtr,   /* rgbValue */
              SQLLEN            BufferLength,        /* cbValueMax */
              SQLLEN            *StrLen_or_IndPtr);  /* pcbValue */
```

Function arguments

Table 10. SQLBindParameter arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement Handle
SQLUSMALLINT	*ParameterNumber*	input	Parameter marker number, ordered sequentially left to right, starting at 1.

Table 10. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*InputOutputType*	input	The type of parameter. The value of the SQL_DESC_PARAMETER_TYPE field of the IPD is also set to this argument. The supported types are: • SQL_PARAM_INPUT: The parameter marker is associated with an SQL statement that is not a stored procedure CALL; or, it marks an input parameter of the CALLed stored procedure. When the statement is executed, the data for the parameter is sent to the server and as such, the *ParameterValuePtr* buffer must contain valid input data value(s), unless the *StrLen_or_IndPtr* buffer contains SQL_NULL_DATA or SQL_DATA_AT_EXEC (if the value should be sent via SQLParamData() and SQLPutData()). • SQL_PARAM_INPUT_OUTPUT: The parameter marker is associated with an input/output parameter of the CALLed stored procedure. When the statement is executed, the data for the parameter is sent to the server and as such, the *ParameterValuePtr* buffer must contain valid input data value(s), unless the *StrLen_or_IndPtr* buffer contains SQL_NULL_DATA or SQL_DATA_AT_EXEC (if the value should be sent via SQLParamData() and SQLPutData()). • SQL_PARAM_OUTPUT: The parameter marker is associated with an output parameter of the CALLed stored procedure or the return value of the stored procedure. After the statement is executed, data for the output parameter is returned to the application buffer specified by *ParameterValuePtr* and *StrLen_or_IndPtr*, unless both are NULL pointers, in which case the output data is discarded. If an output parameter does not have a return value then *StrLen_or_IndPtr* is set to SQL_NULL_DATA.

Table 10. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*ValueType*	input	C data type of the parameter. The following types are supported: • SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CHAR • SQL_C_CLOB_LOCATOR • SQL_C_DBCHAR • SQL_C_DBCLOB_LOCATOR • SQL_C_DECIMAL_IBM • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_NUMERIC [a] • SQL_C_SBIGINT • SQL_C_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TYPE_TIMESTAMP_EXT • SQL_C_TYPE_TIMESTAMP_EXT_TZ • SQL_C_TINYINT • SQL_C_UBIGINT • SQL_C_UTINYINT • SQL_C_WCHAR Specifying SQL_C_DEFAULT causes data to be transferred from its default C data type to the type indicated in *ParameterType*. **a** Windows 32-bit only

Table 10. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*ParameterType*	input	SQL data type of the parameter. The supported types are: • SQL_BIGINT • SQL_BINARY • SQL_BIT • SQL_BLOB • SQL_BLOB_LOCATOR • SQL_CHAR • SQL_CLOB • SQL_CLOB_LOCATOR • SQL_DBCLOB • SQL_DBCLOB_LOCATOR • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_SMALLINT • SQL_TINYINT • SQL_TYPE_DATE • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_TYPE_TIMESTAMP_WITH_TIMEZONE • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC • SQL_WCHAR • SQL_XML **Note:** SQL_BLOB_LOCATOR, SQL_CLOB_LOCATOR, SQL_DBCLOB_LOCATOR are application related concepts and do not map to a data type for column definition during a CREATE TABLE statement.

Table 10. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLULEN	*ColumnSize*	input	Precision of the corresponding parameter marker. If *ParameterType* denotes: • A binary or single byte character string (for example, SQL_CHAR, SQL_BLOB), this is the maximum length in bytes for this parameter marker. • A double byte character string (for example, SQL_GRAPHIC), this is the maximum length in double-byte characters for this parameter. • SQL_DECIMAL, SQL_NUMERIC, this is the maximum decimal precision. • An XML value (SQL_XML) for an external routine argument, this is the maximum length in bytes, n, of the declared XML AS CLOB(n) argument. For all other parameters of type SQL_XML, this argument is ignored. • Otherwise, this argument is ignored.
SQLSMALLINT	*DecimalDigits*	input	If *ParameterType* is SQL_DECIMAL or SQL_NUMERIC, *DecimalDigits* represents the scale of the corresponding parameter and sets the SQL_DESC_SCALE field of the IPD. If *ParameterType* is SQL_TYPE_TIMESTAMP or SQL_TYPE_TIME, *Decimal Digits* represents the precision of the corresponding parameter and sets the SQL_DESC_PRECISION field of the IPD. The precision of a time timestamp value is the number of digits to the right of the decimal point in the string representation of a time or timestamp (for example, the scale of yyyy-mm-dd hh:mm:ss.fff is 3). Other than for the *ParameterType* values mentioned here, *DecimalDigits* is ignored.

Table 10. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLPOINTER	*ParameterValuePtr*	input (deferred), output (deferred), or both	• On input (*InputOutputType* set to SQL_PARAM_INPUT, or SQL_PARAM_INPUT_OUTPUT): At execution time, if *StrLen_or_IndPtr* does not contain SQL_NULL_DATA or SQL_DATA_AT_EXEC, then *ParameterValuePtr* points to a buffer that contains the actual data for the parameter. If *StrLen_or_IndPtr* contains SQL_DATA_AT_EXEC, then *ParameterValuePtr* is an application-defined 32-bit value that is associated with this parameter. This 32-bit value is returned to the application via a subsequent SQLParamData() call. If SQLParamOptions() is called to specify multiple values for the parameter, then *ParameterValuePtr* is a pointer to a input buffer array of *BufferLength* bytes. • On output (*InputOutputType* set to SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT): *ParameterValuePtr* points to the buffer where the output parameter value of the stored procedure will be stored. If *InputOutputType* is set to SQL_PARAM_OUTPUT, and both *ParameterValuePtr* and *StrLen_or_IndPtr* are NULL pointers, then the output parameter value or the return value from the stored procedure call is discarded.

Table 10. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLLEN	*BufferLength*	input	For character and binary data, *BufferLength* specifies the length of the *ParameterValuePtr* buffer (if is treated as a single element) or the length of each element in the *ParameterValuePtr* array (if the application calls SQLParamOptions() to specify multiple values for each parameter). For non-character and non-binary data, this argument is ignored -- the length of the *ParameterValuePtr* buffer (if it is a single element) or the length of each element in the *ParameterValuePtr* array (if SQLParamOptions() is used to specify an array of values for each parameter) is assumed to be the length associated with the C data type. For output parameters, *BufferLength* is used to determine whether to truncate character or binary output data in the following manner: • For character data, if the number of bytes available to return is greater than or equal to *BufferLength*, the data in *ParameterValuePtr* is truncated to *BufferLength-1* bytes and is null-terminated (unless null-termination has been turned off). • For binary data, if the number of bytes available to return is greater than *BufferLength*, the data in *ParameterValuePtr* is truncated to *BufferLength* bytes.

Table 10. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLLEN *	*StrLen_or_IndPtr*	input (deferred), output (deferred), or both	If this is an input or input/output parameter: This is the pointer to the location which contains (when the statement is executed) the length of the parameter marker value stored at *ParameterValuePtr*. To specify a null value for a parameter marker, this storage location must contain SQL_NULL_DATA. If *ValueType* is SQL_C_CHAR, this storage location must contain either the exact length of the data stored at *ParameterValuePtr*, or SQL_NTS if the contents at *ParameterValuePtr* is null-terminated. If *ValueType* indicates character data (explicitly, or implicitly using SQL_C_DEFAULT), and this pointer is set to NULL, it is assumed that the application will always provide a null-terminated string in *ParameterValuePtr*. This also implies that this parameter marker will never have a null value. If *ParameterType* denotes a graphic data type and the *ValueType* is SQL_C_CHAR, the pointer to *StrLen_or_IndPtr* can never be NULL and the contents of *StrLen_or_IndPtr* can never hold SQL_NTS. In general for graphic data types, this length should be the number of octets that the double byte data occupies; therefore, the length should always be a multiple of 2. In fact, if the length is odd, then an error will occur when the statement is executed. When SQLExecute() or SQLExecDirect() is called, and *StrLen_or_IndPtr* points to a value of SQL_DATA_AT_EXEC, the data for the parameter will be sent with SQLPutData(). This parameter is referred to as a **data-at-execution** parameter. When SQLBindParameter() or SQLExtendedBind() method is called through after setting the SQL_ATTR_EXTENDED_INDICATORS attribute, the *StrLen_or_IndPtr* argument allows SQL_UNASSIGNED and SQL_DEFAULT_PARAM constant to pass through the method.

Table 10. SQLBindParameter arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER *	StrLen_or_IndPtr *(cont)*	input (deferred), output (deferred), or both	If SQLSetStmtAttr() is used with the SQL_ATTR_PARAMSET_SIZE attribute to specify multiple values for each parameter, *StrLen_or_IndPtr* points to an array of SQLINTEGER values where each of the elements can be the number of bytes in the corresponding *ParameterValuePtr* element (excluding the null-terminator), or SQL_NULL_DATA. The *StrLen_or_IndPtr* represents the size of the parameter. If you have an output parameter, *StrLen_or_IndPtr* is a memory address (a pointer) to an SQLINTEGER and the value will contain either: • The length of the buffer (minus the NULL terminator). • -1 (SQL_NULL_DATA), which means that the value is NULL, and you can ignore the actual value. • -4 (SQL_NO_TOTAL), which is only used for LOB type of data, and is used to indicate that the number of bytes available to return cannot be determined.

Usage

SQLBindParameter() extends the capability of the deprecated SQLSetParam() function, by providing a method of:
- Specifying whether a parameter is input, input / output, or output, necessary for proper handling of parameters for stored procedures.
- Specifying an array of input parameter values when SQLSetStmtAttr() with the SQL_ATTR_PARAMSET_SIZE attribute is used in conjunction with SQLBindParameter().

This function can be called before SQLPrepare() if the data types and lengths of the target columns in the WHERE or UPDATE clause, or the parameters for the stored procedure are known. Otherwise, you can obtain the attributes of the target columns or stored procedure parameters after the statement is prepared using `SQLDescribeParam()`, and then bind the parameter markers.

Parameter markers are referenced by number (*ParameterNumber*) and are numbered sequentially from left to right, starting at 1.

The C buffer data type given by *ValueType* must be compatible with the SQL data type indicated by *ParameterType*, or an error will occur.

All parameters bound by this function remain in effect until one of the following event takes place:
- SQLFreeStmt() is called with the SQL_RESET_PARAMS option, or
- SQLFreeHandle() is called with *HandleType* set to SQL_HANDLE_STMT, or SQLFreeStmt() is called with the SQL_DROP option, or
- SQLBindParameter() is called again for the same *ParameterNumber*, or

- SQLSetDescField() is called, with the associated APD descriptor handle, to set SQL_DESC_COUNT in the header field of the APD to zero (0).

A parameter can only be bound to either a file or a storage location, not both. The most recent parameter binding function call determines the bind that is in effect.

Parameter type

The *InputOutputType* argument specifies the type of the parameter. All parameters in the SQL statements that do not call procedures are input parameters. Parameters in stored procedure calls can be input, input/output, or output parameters. Even though the DB2 stored procedure argument convention typically implies that all procedure arguments are input/output, the application programmer can still choose to specify more exactly the input or output nature on the SQLBindParameter() to follow a more rigorous coding style.

- If an application cannot determine the type of a parameter in a procedure call, set *InputOutputType* to SQL_PARAM_INPUT; if the data source returns a value for the parameter, CLI discards it.
- If an application has marked a parameter as SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT and the data source does not return a value, CLI sets the *StrLen_or_IndPtr* buffer to SQL_NULL_DATA.
- If an application marks a parameter as SQL_PARAM_OUTPUT, data for the parameter is returned to the application after the CALL statement has been processed. If the *ParameterValuePtr* and *StrLen_or_IndPtr* arguments are both null pointers, CLI discards the output value. If the data source does not return a value for an output parameter, CLI sets the *StrLen_or_IndPtr* buffer to SQL_NULL_DATA.
- For this function, *ParameterValuePtr* and *StrLen_or_IndPtr* are deferred arguments. In the case where *InputOutputType* is set to SQL_PARAM_INPUT or SQL_PARAM_INPUT_OUTPUT, the storage locations must be valid and contain input data values when the statement is executed. This means either keeping the SQLExecDirect() or SQLExecute() call in the same procedure scope as the SQLBindParameter() calls, or, these storage locations must be dynamically allocated or statically / globally declared.

  Similarly, if *InputOutputType* is set to SQL_PARAM_OUTPUT or SQL_PARAM_INPUT_OUTPUT, the *ParameterValuePtr* and *StrLen_or_IndPtr* buffer locations must remain valid until the CALL statement has been executed.

ParameterValuePtr and StrLen_or_IndPtr arguments

ParameterValuePtr and *StrLen_or_IndPtr* are deferred arguments, so the storage locations they point to must be valid and contain input data values when the statement is executed. This means either keeping the SQLExecDirect() or SQLExecute() call in the same application function scope as the SQLBindParameter() calls, or dynamically allocating or statically or globally declaring these storage locations.

Since the data in the variables referenced by *ParameterValuePtr* and *StrLen_or_IndPtr* is not verified until the statement is executed, data content or format errors are not detected or reported until SQLExecute() or SQLExecDirect() is called.

An application can pass the value for a parameter either in the *ParameterValuePtr* buffer or with one or more calls to SQLPutData(). In the latter case, these parameters are data-at-execution parameters. The application informs CLI of a

data-at-execution parameter by placing the SQL_DATA_AT_EXEC value in the buffer pointed to by *StrLen_or_IndPtr*. It sets the *ParameterValuePtr* input argument to a 32-bit value which will be returned on a subsequent SQLParamData() call and can be used to identify the parameter position.

When SQLBindParameter() is used to bind an application variable to an output parameter for a stored procedure, CLI can provide some performance enhancement if the *ParameterValuePtr* buffer is placed consecutively in memory after the *StrLen_or_IndPtr* buffer. For example:

```
struct {  SQLINTEGER  StrLen_or_IndPtr;
          SQLCHAR     ParameterValuePtr[MAX_BUFFER];
       } column;
```

BufferLength argument

For character and binary C data, the *BufferLength* argument specifies the length of the *ParameterValuePtr* buffer if it is a single element; or, if the application calls SQLSetStmtAttr() with the SQL_ATTR_PARAMSET_SIZE attribute to specify multiple values for each parameter, *BufferLength* is the length of *each* element in the *ParameterValuePtr* array, including the null-terminator. If the application specifies multiple values, *BufferLength* is used to determine the location of values in the *ParameterValuePtr* array. For all other types of C data, the *BufferLength* argument is ignored.

ColumnSize argument

When actual size of the target column or output parameter is not known, the application can specify 0 for the length of the column. (*ColumnSize* set to 0).

If the column's data type is of fixed-length, the CLI driver will base the length from the data type itself. However, setting *ColumnSize* to 0 means different things when the data type is of type character, binary string or large object:

Input parameter
A 0 *ColumnSize* means that CLI will use the maximum length for the SQL type provided as the size of the column or stored procedure parameter. CLI will perform any necessary conversions using this size.

Output parameter (stored procedures only)
A 0 *ColumnSize* means that CLI will use *BufferLength* as the parameter's size. Note that this means that the stored procedure must not return more than *BufferLength* bytes of data or a truncation error will occur.

For Input-output parameter (store procedures only)
A 0 *ColumnSize* means that CLI will set both the input and output to *BufferLength* as the target parameter. This means that the input data will be converted to this new size if necessary before being sent to the stored procedure and at most *BufferLength* bytes of data are expected to be returned.

Setting *ColumnSize* to 0 is not recommended unless it is required; it causes CLI to perform costly checking for the length of the data at run time.

Descriptors

How a parameter is bound is determined by fields of the APD and IPD. The arguments in `SQLBindParameter()` are used to set those descriptor fields. The fields

can also be set by the SQLSetDescField() functions, although SQLBindParameter() is more efficient to use because the application does not have to obtain a descriptor handle to call SQLBindParameter().

Note: Calling SQLBindParameter() for one statement can affect other statements. This occurs when the APD associated with the statement is explicitly allocated and is also associated with other statements. Because SQLBindParameter() modifies the fields of the APD, the modifications apply to all statements with which this descriptor is associated. If this is not the required behavior, the application should dissociate the descriptor from the other statements before calling SQLBindParameter().

Conceptually, SQLBindParameter() performs the following steps in sequence:

- Calls SQLGetStmtAttr() to obtain the APD handle.
- Calls SQLGetDescField() to get the SQL_DESC_COUNT header field from the APD, and if the value of the *ParameterNumber* argument exceeds the value of SQL_DESC_COUNT, calls SQLSetDescField() to increase the value of SQL_DESC_COUNT to *ParameterNumber*.
- Calls SQLSetDescField() multiple times to assign values to the following fields of the APD:
  - Sets SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE to the value of *ValueType*, except that if *ValueType* is one of the concise identifiers of a datetime, it sets SQL_DESC_TYPE to SQL_DATETIME, sets SQL_DESC_CONCISE_TYPE to the concise identifier, and sets SQL_DESC_DATETIME_INTERVAL_CODE to the corresponding datetime subcode.
  - Sets the SQL_DESC_DATA_PTR field to the value of *ParameterValue*.
  - Sets the SQL_DESC_OCTET_LENGTH_PTR field to the value of *StrLen_or_Ind*.
  - Sets the SQL_DESC_INDICATOR_PTR field also to the value of *StrLen_or_Ind*.

  The *StrLen_or_Ind* parameter specifies both the indicator information and the length for the parameter value.
- Calls SQLGetStmtAttr() to obtain the IPD handle.
- Calls SQLGetDescField() to get the IPD's SQL_DESC_COUNT field, and if the value of the *ParameterNumber* argument exceeds the value of SQL_DESC_COUNT, calls SQLSetDescField() to increase the value of SQL_DESC_COUNT to *ParameterNumber*.
- Calls SQLSetDescField() multiple times to assign values to the following fields of the IPD:
  - Sets SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE to the value of *ParameterType*, except that if *ParameterType* is one of the concise identifiers of a datetime, it sets SQL_DESC_TYPE to SQL_DATETIME, sets SQL_DESC_CONCISE_TYPE to the concise identifier, and sets SQL_DESC_DATETIME_INTERVAL_CODE to the corresponding datetime subcode.
  - Sets one or more of SQL_DESC_LENGTH, SQL_DESC_PRECISION, and SQL_DESC_SCALE as appropriate for *ParameterType*.

If the call to SQLBindParameter() fails, the content of the descriptor fields that it would have set in the APD are undefined, and the SQL_DESC_COUNT field of the APD is unchanged. In addition, the SQL_DESC_LENGTH, SQL_DESC_PRECISION, SQL_DESC_SCALE, and SQL_DESC_TYPE fields of the appropriate record in the IPD are undefined and the SQL_DESC_COUNT field of the IPD is unchanged.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 11. SQLBindParameter SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The conversion from the data value identified by the *ValueType* argument to the data type identified by the *ParameterType* argument is not a meaningful conversion. (For example, conversion from SQL_C_TYPE_DATE to SQL_DOUBLE.)
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY003	Program type out of range.	The value specified by the argument *ParameterNumber* not a valid data type or SQL_C_DEFAULT.
HY004	SQL data type out of range.	The value specified for the argument *ParameterType* is not a valid SQL data type.
HY009	Invalid argument value.	The argument *ParameterValuePtr* was a null pointer and the argument *StrLen_or_IndPtr* was a null pointer, and *InputOutputType* is not SQL_PARAM_OUTPUT.
HY010	Function sequence error.	Function was called after SQLExecute() or SQLExecDirect() had returned SQL_NEED_DATA, but data has not been sent for all *data-at-execution* parameters.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY021	Inconsistent descriptor information	The descriptor information checked during a consistency check was not consistent.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* was less than 0.
HY093	Invalid parameter number.	The value specified for the argument *ValueType* was less than 1 or greater than the maximum number of parameters supported by the server.
HY094	Invalid scale value.	The value specified for *ParameterType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified for *DecimalDigits* was less than 0 or greater than the value for the argument *ParamDef* (precision). The value specified for *ParameterType* was SQL_C_TYPE_TIMESTAMP and the value for *ParameterType* was either SQL_CHAR or SQL_VARCHAR and the value for *DecimalDigits* was less than 0 or greater than 9. The value specified for *ParameterType* was SQL_C_TIMESTAMP_EXT and the value for *ParameterType* was either SQL_CHAR or SQL_VARCHAR and the value for *DecimalDigits* was less than 0 or greater than 12.

Table 11. SQLBindParameter SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY104	Invalid precision value.	The value specified for *ParameterType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified for *ParamDef* was less than 1.
HY105	Invalid parameter type.	*InputOutputType* is not one of SQL_PARAM_INPUT, SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT.
HYC00	Driver not capable.	CLI or data source does not support the conversion specified by the combination of the value specified for the argument *ValueType* and the value specified for the argument *ParameterType*. The value specified for the argument *ParameterType* is not supported by either CLI or the data source.

Restrictions

`SQLBindParameter()` replaces the deprecated `SQLSetParam()` API in CLI V5 and later, and ODBC 2.0 and later.

An additional value for *StrLen_or_IndPtr*, SQL_DEFAULT_PARAM, was introduced in ODBC 2.0, to indicate that the procedure is to use the default value of a parameter, rather than a value sent from the application. Since DB2 stored procedure arguments do not support default values, specification of this value for *StrLen_or_IndPtr* argument will result in an error when the CALL statement is executed since the SQL_DEFAULT_PARAM value will be considered an invalid length.

ODBC 2.0 also introduced the SQL_LEN_DATA_AT_EXEC(*length*) macro to be used with the *StrLen_or_IndPtr* argument. The macro is used to specify the sum total length of the entire data that would be sent for character or binary C data via the subsequent SQLPutData() calls. Since the DB2 ODBC driver does not need this information, the macro is not needed. An ODBC application calls SQLGetInfo() with the SQL_NEED_LONG_DATA_LEN option to check if the driver needs this information. The DB2 ODBC driver will return 'N' to indicate that this information is not needed by SQLPutData().

Example

```
SQLSMALLINT parameter1 = 0;

/* ... */

cliRC = SQLBindParameter(hstmt,
                         1,
                         SQL_PARAM_INPUT,
                         SQL_C_SHORT,
                         SQL_SMALLINT,
                         0,
                         0,
                         &parameter1,
                         0,
                         NULL);
```

SQLBrowseConnect function (CLI) - Get required attributes to connect to data source

Supports an iterative method of discovering and enumerating the attributes and attribute values required to connect to a data source.

Specification:

- **CLI** 5.0
- **ODBC** 1

Each call to SQLBrowseConnect() returns successive levels of attributes and attribute values. When all levels have been enumerated, a connection to the data source is completed and a complete connection string is returned by SQLBrowseConnect(). A return code of SQL_SUCCESS or SQL_SUCCESS_WITH_INFO indicates that all connection information has been specified and the application is now connected to the data source.

Unicode Equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLBrowseConnectW(). Refer to "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN  SQLBrowseConnect (
             SQLHDBC      ConnectionHandle,              /* hdbc */
             SQLCHAR      *InConnectionString,           /* *szConnStrIn */
             SQLSMALLINT  InConnectionStringLength,      /* dbConnStrIn */
             SQLCHAR      *OutConnectionString,          /* *szConnStrOut */
             SQLSMALLINT  OutConnectionStringCapacity,   /* dbConnStrOutMax */
             SQLSMALLINT  *OutConnectionStringLengthPtr); /* *pcbConnStrOut */
```

Function Arguments

Table 12. SQLBrowseConnect arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection handle.
SQLCHAR *	*InConnectionString*	input	Browse request connection string (see InConnectionString argument).
SQLSMALLINT	*InConnectionStringLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store **InConnectionString*.
SQLCHAR *	*OutConnectionString*	output	Pointer to a buffer in which to return the browse result connection string (see OutConnectionString argument).
SQLSMALLINT	*OutConnectionString Capacity*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the **OutConnectionString* buffer.
SQLSMALLINT *	*OutConnectionString LengthPtr*	output	The total number of elements (excluding the null termination character) available to return in **OutConnectionString*. If the number of elements available to return is greater than or equal to *OutConnectionStringCapacity*, the connection string in **OutConnectionString* is truncated to *OutConnectionStringCapacity* minus the length of a null termination character.

Usage

SQLBrowseConnect() requires an allocated connection. If SQLBrowseConnect() returns SQL_ERROR, outstanding connection information is discarded, and the connection is returned to an unconnected state.

When SQLBrowseConnect() is called for the first time on a connection, the browse request connection string must contain the DSN keyword.

On each call to SQLBrowseConnect(), the application specifies the connection attribute values in the browse request connection string. CLI returns successive levels of attributes and attribute values in the browse result connection string; it returns SQL_NEED_DATA as long as there are connection attributes that have not yet been enumerated in the browse request connection string. The application uses the contents of the browse result connection string to build the browse request connection string for the next call to SQLBrowseConnect(). All mandatory attributes (those not preceded by an asterisk in the *OutConnectionString* argument) must be included in the next call to SQLBrowseConnect(). Note that the application cannot simply copy the entire content of previous browse result connection strings when building the current browse request connection string; that is, it cannot specify different values for attributes set in previous levels.

When all levels of connection and their associated attributes have been enumerated, CLI returns SQL_SUCCESS, the connection to the data source is complete, and a complete connection string is returned to the application. The connection string is suitable to use as an argument for SQLDriverConnect() in conjunction with the SQL_DRIVER_NOPROMPT option to establish another connection. The complete connection string cannot be used in another call to SQLBrowseConnect(), however; if SQLBrowseConnect() were called again, the entire sequence of calls would have to be repeated.

SQLBrowseConnect() also returns SQL_NEED_DATA if there are recoverable, nonfatal errors during the browse process, for example, an invalid password supplied by the application or an invalid attribute keyword supplied by the application. When SQL_NEED_DATA is returned and the browse result connection string is unchanged, an error has occurred and the application can call SQLGetDiagRec() to return the SQLSTATE for browse-time errors. This permits the application to correct the attribute and continue the browse.

An application can terminate the browse process at any time by calling SQLDisconnect(). CLI will terminate any outstanding connection information and return the connection to an unconnected state.

InConnectionString argument

A browse request connection string has the following syntax:

connection-string ::= attribute[] | attribute: connection-string

attribute ::= attribute-keyword=attribute-value
| DRIVER=[{]attribute-value[}]

attribute-keyword ::= DSN | UID | PWD | NEWPWD
| driver-defined-attribute-keyword

attribute-value ::= character-string
driver-defined-attribute-keyword ::= identifier

where
- character-string has zero or more SQLCHAR or SQLWCHAR elements
- identifier has one or more SQLCHAR or SQLWCHAR elements
- attribute-keyword is case insensitive
- attribute-value might be case sensitive
- the value of the **DSN** keyword does not consist solely of blanks
- **NEWPWD** is used as part of a change password request. The application can either specify the new string to use, for example, `NEWPWD=anewpass;` or specify `NEWPWD=;` and rely on a dialog box generated by the CLI driver to prompt for the new password

Because of connection string and initialization file grammar, keywords and attribute values that contain the characters **[]{}(),;?*=!@** should be avoided. Because of the grammar in the system information, keywords and data source names cannot contain the backslash (\) character. For CLI Version 2, braces are required around the **DRIVER** keyword.

If any keywords are repeated in the browse request connection string, CLI uses the value associated with the first occurrence of the keyword. If the **DSN** and **DRIVER** keywords are included in the same browse request connection string, CLI uses whichever keyword appears first.

OutConnectionString argument

The browse result connection string is a list of connection attributes. A connection attribute consists of an attribute keyword and a corresponding attribute value. The browse result connection string has the following syntax:

connection-string ::= attribute[;] | attribute; connection-string

attribute ::= [*]attribute-keyword=attribute-value
attribute-keyword ::= ODBC-attribute-keyword
| driver-defined-attribute-keyword

ODBC-attribute-keyword = {UID | PWD}[:localized-identifier]
driver-defined-attribute-keyword ::= identifier[:localized-identifier]

attribute-value ::= {attribute-value-list} | ?
(The braces are literal; they are returned by CLI.)
attribute-value-list ::= character-string [:localized-character
string] | character-string [:localized-character string], attribute-value-list

where
- character-string and localized-character string have zero or more SQLCHAR or SQLWCHAR elements
- identifier and localized-identifier have one or more elements; attribute-keyword is case insensitive
- attribute-value might be case sensitive

Because of connection string and initialization file grammar, keywords, localized identifiers, and attribute values that contain the characters **[]{}(),;?*=!@** should be avoided. Because of the grammar in the system information, keywords and data source names cannot contain the backslash (\) character.

The browse result connection string syntax is used according to the following semantic rules:

- If an asterisk (*) precedes an attribute-keyword, the attribute is optional, and can be omitted in the next call to SQLBrowseConnect().
- The attribute keywords **UID** and **PWD** have the same meaning as defined in SQLDriverConnect().
- When connecting to a DB2 database, only **DSN**, **UID** and **PWD** are required. Other keywords can be specified but do not affect the connection.
- ODBC-attribute-keywords and driver-defined-attribute-keywords include a localized or user-friendly version of the keyword. This might be used by applications as a label in a dialog box. However, **UID**, **PWD**, or the identifier alone must be used when passing a browse request string to CLI.
- The {attribute-value-list} is an enumeration of actual values valid for the corresponding attribute-keyword. Note that the braces ({}) do not indicate a list of choices; they are returned by CLI. For example, it might be a list of server names or a list of database names.
- If the attribute-value is a single question mark (?), a single value corresponds to the attribute-keyword. For example, UID=JohnS; PWD=Sesame.
- Each call to SQLBrowseConnect() returns only the information required to satisfy the next level of the connection process. CLI associates state information with the connection handle so that the context can always be determined on each call.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 13. SQLBrowseConnect SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The buffer **OutConnectionString* was not large enough to return entire browse result connection string, so the string was truncated. The buffer **OutConnectionStringLengthPtr* contains the length of the untruncated browse result connection string. (Function returns SQL_SUCCESS_WITH_INFO.)
01S00	Invalid connection string attribute.	An invalid attribute keyword was specified in the browse request connection string (*InConnectionString*). (Function returns SQL_NEED_DATA.) An attribute keyword was specified in the browse request connection string (*InConnectionString*) that does not apply to the current connection level. (Function returns SQL_NEED_DATA.)
01S02	Option value changed.	CLI did not support the specified value of the *ValuePtr* argument in `SQLSetConnectAttr()` and substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)
08001	Unable to connect to data source.	CLI was unable to establish a connection with the data source.
08002	Connection in use.	The specified connection had already been used to establish a connection with a data source and the connection was open.

Table 13. SQLBrowseConnect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
08004	The application server rejected establishment of the connection.	The data source rejected the establishment of the connection for implementation defined reasons.
08S01	Communication link failure.	The communication link between CLI and the data source to which it was trying to connect failed before the function completed processing.
28000	Invalid authorization specification.	Either the user identifier or the authorization string or both as specified in the browse request connection string (*InConnectionString*) violated restrictions defined by the data source.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for argument *InConnectionStringLength* was less than 0 and was not equal to SQL_NTS. The value specified for argument *OutConnectionStringCapacity* was less than 0.

Restrictions

None.

Example

```
SQLCHAR connInStr[255]; /* browse request connection string */
SQLCHAR outStr[1025]; /* browse result connection string*/

/* ... */

cliRC = SQL_NEED_DATA;
while (cliRC == SQL_NEED_DATA)
{
  /* get required attributes to connect to data source */
  cliRC = SQLBrowseConnect(hdbc,
                           connInStr,
                           SQL_NTS,
                           outStr,
                           sizeof(outStr),
                           &indicator);
  DBC_HANDLE_CHECK(hdbc, cliRC);

  printf("  So far, have connected %d times to database %s\n",
         count++, db1Alias);
  printf("  Resulting connection string:

  /* if inadequate connection information was provided, exit
     the program */
  if (cliRC == SQL_NEED_DATA)
  {
    printf("  You can provide other connection information "
```

```
            "here by setting connInStr\n");
      break;
    }

    /* if the connection was successful, output confirmation */
    if (cliRC == SQL_SUCCESS)
    {
      printf("  Connected to the database
    }
  }
```

SQLBulkOperations function (CLI) - Add, update, delete, or fetch a set of rows

Adds, updates, deletes, or fetches a set of rows on a keyset-driven cursor.

Specification:

- **CLI** 6.0
- **ODBC** 3.0

SQLBulkOperations() is used to perform the following operations on a keyset-driven cursor:
- Add new rows
- Update a set of rows where each row is identified by a bookmark
- Delete a set of rows where each row is identified by a bookmark
- Fetch a set of rows where each row is identified by a bookmark

Syntax

```
SQLRETURN   SQLBulkOperations    (
                SQLHSTMT          StatementHandle,
                SQLSMALLINT       Operation);
```

Function arguments

Table 14. SQLBulkOperations arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	Statement handle.
SQLSMALLINT	*Operation*	Input	Operation to perform: • SQL_ADD • SQL_UPDATE_BY_BOOKMARK • SQL_DELETE_BY_BOOKMARK • SQL_FETCH_BY_BOOKMARK

Usage

An application uses SQLBulkOperations() to perform the following operations on the base table or view that corresponds to the current query in a keyset-driven cursor:
- Add new rows
- Update a set of rows where each row is identified by a bookmark
- Delete a set of rows where each row is identified by a bookmark
- Fetch a set of rows where each row is identified by a bookmark

A generic application should first ensure that the required bulk operation is supported. To do so, it can call SQLGetInfo() with an *InfoType* of SQL_DYNAMIC_CURSOR_ATTRIBUTES1 and

SQL_DYNAMIC_CURSOR_ATTRIBUTES2 (for example, to see if SQL_CA1_BULK_UPDATE_BY_BOOKMARK is returned)

After a call to SQLBulkOperations(), the block cursor position is undefined. The application has to call SQLFetchScroll() to set the cursor position. An application should only call SQLFetchScroll() with a *FetchOrientation* argument of SQL_FETCH_FIRST, SQL_FETCH_LAST, SQL_FETCH_ABSOLUTE, or SQL_FETCH_BOOKMARK. The cursor position is undefined if the application calls SQLFetch(), or SQLFetchScroll() with a *FetchOrientation* argument of SQL_FETCH_PRIOR, SQL_FETCH_NEXT, or SQL_FETCH_RELATIVE.

A column can be ignored in bulk operations (calls to SQLBulkOperations()). To do so, call SQLBindCol() and set the column length/indicator buffer (*StrLen_or_IndPtr*) to SQL_COLUMN_IGNORE. This does not apply to SQL_DELETE_BY_BOOKMARK bulk operation.

It is not necessary for the application to set the SQL_ATTR_ROW_OPERATION_PTR statement attribute when calling SQLBulkOperations() because rows cannot be ignored when performing bulk operations with this function.

The buffer pointed to by the SQL_ATTR_ROWS_FETCHED_PTR statement attribute contains the number of rows affected by a call to SQLBulkOperations().

When the *Operation* argument is SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the select-list of the query specification associated with the cursor contains more than one reference to the same column, an error is generated.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 15. SQLBulkOperations SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The *Operation* argument was SQL_FETCH_BY_BOOKMARK, and string or binary data returned for a column or columns with a data type of SQL_C_CHAR or SQL_C_BINARY resulted in the truncation of non-blank character or non-NULL binary data.
01S07	Invalid conversion.	The *Operation* argument was SQL_FETCH_BY_BOOKMARK, the data type of the application buffer was not SQL_C_CHAR or SQL_C_BINARY, and the data returned to application buffers for one or more columns was truncated. (For numeric C data types, the fractional part of the number was truncated. For time and timestamp data types, the fractional portion of the time was truncated.) (Function returns SQL_SUCCESS_WITH_INFO.)

Table 15. SQLBulkOperations SQLSTATEs (continued)

SQLSTATE	Description	Explanation
07006	Restricted data type attribute violation.	The *Operation* argument was SQL_FETCH_BY_BOOKMARK, and the data value of a column in the result set could not be converted to the data type specified by the *TargetType* argument in the call to `SQLBindCol()`. The *Operation* argument was SQL_UPDATE_BY_BOOKMARK or SQL_ADD, and the data value in the application buffers could not be converted to the data type of a column in the result set.
07009	Invalid descriptor index.	The argument *Operation* was SQL_ADD and a column was bound with a column number greater than the number of columns in the result set, or the column number was less than 0.
21S02	Degree of derived table does not match column list.	The argument *Operation* was SQL_UPDATE_BY_BOOKMARK; and no columns were updatable because all columns were either unbound, read-only, or the value in the bound length/indicator buffer was SQL_COLUMN_IGNORE.
22001	String data right truncation.	The assignment of a character or binary value to a column in the result set resulted in the truncation of non-blank (for characters) or non-null (for binary) characters or bytes.
22003	Numeric value out of range.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of a numeric value to a column in the result set caused the whole (as opposed to fractional) part of the number to be truncated. The argument *Operation* was SQL_FETCH_BY_BOOKMARK, and returning the numeric value for one or more bound columns would have caused a loss of significant digits.
22007	Invalid datetime format.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of a date or timestamp value to a column in the result set caused the year, month, or day field to be out of range. The argument *Operation* was SQL_FETCH_BY_BOOKMARK, and returning the date or timestamp value for one or more bound columns would have caused the year, month, or day field to be out of range.
22008	Date/time field overflow.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the performance of datetime arithmetic on data being sent to a column in the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result being outside the permissible range of values for the field, or being invalid based on the natural rules for datetimes based on the Gregorian calendar. The *Operation* argument was SQL_FETCH_BY_BOOKMARK, and the performance of datetime arithmetic on data being retrieved from the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result being outside the permissible range of values for the field, or being invalid based on the natural rules for datetimes based on the Gregorian calendar.

Table 15. SQLBulkOperations SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22018	Invalid character value for cast specification.	The *Operation* argument was SQL_FETCH_BY_BOOKMARK; the C type was an exact or approximate numeric or datetime data type; the SQL type of the column was a character data type; and the value in the column was not a valid literal of the bound C type. The argument *Operation* was SQL_ADD or SQL_UPDATE_BY_BOOKMARK; the SQL type was an exact or approximate numeric or datetime data type; the C type was SQL_C_CHAR; and the value in the column was not a valid literal of the bound SQL type.
23000	Integrity constraint violation.	The *Operation* argument was SQL_ADD, SQL_DELETE_BY_BOOKMARK, or SQL_UPDATE_BY_BOOKMARK, and an integrity constraint was violated. The *Operation* argument was SQL_ADD, and a column that was not bound is defined as NOT NULL and has no default. The *Operation* argument was SQL_ADD, the length specified in the bound *StrLen_or_IndPtr* buffer was SQL_COLUMN_IGNORE, and the column did not have a default value.
24000	Invalid cursor state.	The *StatementHandle* was in an executed state but no result set was associated with the *StatementHandle*. `SQLFetch()` or `SQLFetchScroll()` was not called by the application after `SQLExecute()` or `SQLExecDirect()`.
40001	Serialization failure.	The transaction was rolled back due to a resource deadlock with another transaction.
40003	Statement completion unknown.	The associated connection failed during the execution of this function and the state of the transaction cannot be determined.
42000	Syntax error or access violation.	CLI was unable to lock the row as needed to perform the operation requested in the *Operation* argument.
44000	WITH CHECK OPTION violation.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the insert or update was performed on a viewed table or a table derived from the viewed table which was created by specifying WITH CHECK OPTION, such that one or more rows affected by the insert or update will no longer be present in the viewed table.
HY000	General error.	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation error.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.

Table 15. SQLBulkOperations SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called For the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY011	Operation invalid at this time.	The SQL_ATTR_ROW_STATUS_PTR statement attribute was set between calls to `SQLFetch()` or `SQLFetchScroll()` and SQLBulkOperations.
HY013	Unexpected memory handling error.	CLI was unable to access memory required to support execution or completion of this function.
HY090	Invalid string or buffer length.	The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, a data value was a null pointer, and the column length value was not 0, SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, a data value was not a null pointer; the C data type was SQL_C_BINARY or SQL_C_CHAR; and the column length value was less than 0, but not equal to SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NTS, or SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. The value in a length/indicator buffer was SQL_DATA_AT_EXEC; the SQL type was either SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long data type; and the SQL_NEED_LONG_DATA_LEN information type in `SQLGetInfo()` was “Y”. The *Operation* argument was SQL_ADD, the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and column 0 was bound to a buffer whose length was not equal to the maximum length for the bookmark for this result set. (This length is available in the SQL_DESC_OCTET_LENGTH field of the IRD, and can be obtained by calling `SQLDescribeCol()`, `SQLColAttribute()`, or `SQLGetDescField()`.)

Table 15. SQLBulkOperations SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY092	Invalid attribute identifier.	The value specified for the *Operation* argument was invalid. The *Operation* argument was SQL_ADD, SQL_UPDATE_BY_BOOKMARK, or SQL_DELETE_BY_BOOKMARK, and the SQL_ATTR_CONCURRENCY statement attribute was set to SQL_CONCUR_READ_ONLY. The *Operation* argument was SQL_DELETE_BY_BOOKMARK, SQL_FETCH_BY_BOOKMARK, or SQL_UPDATE_BY_BOOKMARK, and the bookmark column was not bound or the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.
HYC00	Optional feature not implemented.	CLI or data source does not support the operation requested in the *Operation* argument.
HYT00	Timeout expired.	The query timeout period expired before the data source returned the result set. The timeout period is set through `SQLSetStmtAttr()` with an *Attribute* argument of SQL_ATTR_QUERY_TIMEOUT.
HYT01	Connection timeout expired.	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through `SQLSetConnectAttr()`, SQL_ATTR_CONNECTION_TIMEOUT.

Restrictions

None.

SQLCancel function (CLI) - Cancel statement

Facilitates premature termination of the data-at-execution sequence for sending and retrieving long data in pieces. It can also be used to cancel a function called in a different thread.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

Syntax

```
SQLRETURN   SQLCancel       (SQLHSTMT         StatementHandle);  /* hstmt */
```

Function arguments

Table 16. SQLCancel arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle

Usage

After SQLExecDirect() or SQLExecute() returns SQL_NEED_DATA to solicit for values for data-at-execution parameters, SQLCancel() can be used to cancel the

data-at-execution sequence for sending and retrieving long data in pieces. SQLCancel() can be called any time before the final SQLParamData() in the sequence. After the cancellation of this sequence, the application can call SQLExecute() or SQLExecDirect() to re-initiate the data-at-execution sequence.

If no processing is being done on the statement, SQLCancel() has no effect. Applications should not call SQLCancel() to close a cursor, but rather SQLFreeStmt() should be used.

Canceling queries on host databases

To call `SQLCancel()` against a server which does not have native interrupt support (such as DB2 for z/OS®, Version 7 and earlier, and IBM DB2 for IBM i), the `INTERRUPT_ENABLED` option must be set when cataloging the DCS database entry for the server.

When the `INTERRUPT_ENABLED` option is set and `SQLCancel()` is received by the server, the server drops the connection and rolls back the unit of work. The application receives an SQL30081N error indicating that the connection to the server has been terminated. In order for the application to process additional database requests, the application must establish a new connection with the database server.

Canceling asynchronous processing

After an application calls a function asynchronously, it calls the function repeatedly to determine whether it has finished processing. If the function is still processing, it returns SQL_STILL_EXECUTING.

After any call to the function that returns SQL_STILL_EXECUTING, an application can call SQLCancel() to cancel the function. If the cancel request is successful, SQL_SUCCESS is returned. This message does not indicate that the function was actually canceled; it indicates that the cancel request was processed. The application must then continue to call the original function until the return code is not SQL_STILL_EXECUTING. If the function was successfully canceled, the return code is for that function is SQL_ERROR and SQLSTATE HY008 (Operation was Canceled.). If the function succeeded by completing its normal processing, the return code is SQL_SUCCESS or SQL_SUCCESS_WITH_INFO. If the function failed for reasons other than cancellation, the return code is SQL_ERROR and an SQLSTATE other than HY008 (Operation was Canceled.).

Canceling functions in multithread applications

In a multithread application, the application can cancel a function that is running synchronously on a statement. To cancel the function, the application calls SQLCancel() with the same statement handle as that used by the target function, but on a different thread. How the function is canceled depends upon the operating system. The return code of the SQLCancel() call indicates only whether CLI processed the request successfully. Only SQL_SUCCESS or SQL_ERROR can be returned; no SQLSTATEs are returned. If the original function is canceled, it returns SQL_ERROR and SQLSTATE HY008 (Operation was Canceled.).

If an SQL statement is being executed when SQLCancel() is called on another thread to cancel the statement execution, it is possible that the execution succeeds and returns SQL_SUCCESS, while the cancel is also successful. In this case, CLI assumes that the cursor opened by the statement execution is closed by the cancel,

so the application will not be able to use the cursor.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_INVALID_HANDLE
- SQL_ERROR

Note: SQL_SUCCESS means that the cancel request was processed, not that the function call was canceled.

Diagnostics

Table 17. SQLCancel SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY018	Server declined cancel request.	The server declined the cancel request.
HY506	Error closing a file.	An error occurred when closing the temporary file generated by CLI when inserting LOB data in pieces using SQLParamData()/ SQLPutData().

Restrictions

None.

Example

```
/* cancel the SQL_DATA_AT_EXEC state for hstmt */
cliRC = SQLCancel(hstmt);
```

SQLCloseCursor function (CLI) - Close cursor and discard pending results

Closes a cursor that has been opened on a statement and discards pending results.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Syntax

```
SQLRETURN   SQLCloseCursor (SQLHSTMT     StatementHandle);  /* hStmt */
```

Function arguments

Table 18. SQLCloseCursor arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle

Usage

After an application calls SQLCloseCursor(), the application can reopen the cursor later by executing a SELECT statement again with the same or different parameter values. SQLCloseCursor() can be called before a transaction is completed.

SQLCloseCursor() returns SQLSTATE 24000 (Invalid cursor state) if no cursor is open. Calling SQLCloseCursor() is equivalent to calling `SQLFreeStmt()` with the SQL_CLOSE option, with the exception that `SQLFreeStmt()` with SQL_CLOSE has no effect on the application if no cursor is open on the statement, while SQLCloseCursor() returns SQLSTATE 24000 (Invalid cursor state).

The statement attribute SQL_ATTR_CLOSE_BEHAVIOR can be used to indicate whether or not CLI should attempt to release read locks acquired during a cursor's operation when the cursor is closed. If SQL_ATTR_CLOSE_BEHAVIOR is set to SQL_CC_RELEASE then the database manager will attempt to release all read locks (if any) that have been held for the cursor.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 19. SQLCloseCursor SQLSTATEs

SQLSTATE	Description	Explanation
01000	General warning	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
24000	Invalid cursor state.	No cursor was open on the *StatementHandle*. (This is returned only by CLI Version 5 or later.)
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	An asynchronously executing function was called for the *StatementHandle* and was still executing when this function was called. `SQLExecute()` or `SQLExecDirect()` was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

Table 19. SQLCloseCursor SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

```
/* close the cursor */
  cliRC = SQLCloseCursor(hstmt);
```

SQLColAttribute function (CLI) - Return a column attribute

Returns descriptor information for a column in a result set. Descriptor information is returned as a character string, a 32-bit descriptor-dependent value, or an integer value.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLColAttributeW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

In a Windows 64-bit environment, the syntax is as follows:

```
SQLRETURN   SQLColAttribute  (
                SQLHSTMT          StatementHandle,       /* hstmt */
                SQLSMALLINT       ColumnNumber,          /* icol */
                SQLSMALLINT       FieldIdentifier,       /* fDescType */
                SQLPOINTER        CharacterAttributePtr, /* rgbDesc */
                SQLSMALLINT       BufferLength,          /* cbDescMax */
                SQLSMALLINT       *StringLengthPtr,      /* pcbDesc */
                SQLLEN            *NumericAttributePtr); /* pfDesc */
```

The syntax for all other platforms is as follows:

```
SQLRETURN   SQLColAttribute  (
                SQLHSTMT          StatementHandle,       /* hstmt */
                SQLSMALLINT       ColumnNumber,          /* icol */
                SQLSMALLINT       FieldIdentifier,       /* fDescType */
                SQLPOINTER        CharacterAttributePtr, /* rgbDesc */
                SQLSMALLINT       BufferLength,          /* cbDescMax */
                SQLSMALLINT       *StringLengthPtr,      /* pcbDesc */
                SQLPOINTER        NumericAttributePtr);  /* pfDesc */
```

Function arguments

Table 20. SQLColAttribute arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLUSMALLINT	*ColumnNumber*	input	The number of the record in the IRD from which the field value is to be retrieved. This argument corresponds to the column number of result data, ordered sequentially from left to right, starting at 1. Columns can be described in any order. Column 0 can be specified in this argument, but all values except SQL_DESC_TYPE and SQL_DESC_OCTET_LENGTH will return undefined values.
SQLSMALLINT	*FieldIdentifier*	input	The field in row *ColumnNumber* of the IRD that is to be returned (see Table 21 on page 53).
SQLPOINTER	*CharacterAttributePtr*	output	Pointer to a buffer in which to return the value in the *FieldIdentifier* field of the *ColumnNumber* row of the IRD, if the field is a character string. Otherwise, the field is unused.
SQLINTEGER	*BufferLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the **CharacterAttributePtr* buffer, if the field is a character string. Otherwise, the field is ignored.
SQLSMALLINT *	*StringLengthPtr*	output	Pointer to a buffer in which to return the total number of bytes (excluding the byte count of the null termination character for character data) available to return in **CharacterAttributePtr*. For character data, if the number of bytes available to return is greater than or equal to *BufferLength*, the descriptor information in **CharacterAttributePtr* is truncated to *BufferLength* minus the length of a null termination character and is null-terminated by CLI. For all other types of data, the value of *BufferLength* is ignored and CLI assumes the size of **CharacterAttributePtr* is 32 bits.
SQLLEN* (Window 64-bit) or SQLPOINTER	*NumericAttributePtr*	output	Pointer to a buffer in which to return the value in the *FieldIdentifier* field of the *ColumnNumber* row of the IRD, if the field is a numeric descriptor type, such as SQL_DESC_COLUMN_LENGTH. Otherwise, the field is unused.

Usage

SQLColAttribute() returns information either in **NumericAttributePtr* or in **CharacterAttributePtr*. Integer information is returned in **NumericAttributePtr* as a 32-bit, signed value; all other formats of information are returned in **CharacterAttributePtr*. When information is returned in **NumericAttributePtr*, CLI ignores *CharacterAttributePtr*, *BufferLength*, and *StringLengthPtr* When information is returned in **CharacterAttributePtr*, CLI ignores *NumericAttributePtr*.

SQLColAttribute() returns values from the descriptor fields of the IRD. The function is called with a statement handle rather than a descriptor handle. The values returned by SQLColAttribute() for the *FieldIdentifier* values, listed in the following table, can also be retrieved by calling SQLGetDescField() with the appropriate IRD handle.

The currently defined descriptor types, the version of CLI in which they were introduced (perhaps with a different name), and the arguments in which information is returned for them are shown in the following table; it is expected that more descriptor types will be defined to take advantage of different data sources.

CLI must return a value for each of the descriptor types. If a descriptor type does not apply to a data source, then, unless otherwise stated, CLI returns 0 in **StringLengthPtr* or an empty string in **CharacterAttributePtr*.

The following table lists the descriptor types returned by SQLColAttribute().

Table 21. SQLColAttribute arguments

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_AUTO_UNIQUE_VALUE (DB2 CLI v2)	Numeric AttributePtr	Indicates if the column data type is an auto increment data type. SQL_FALSE is returned in *NumericAttributePtr* for all DB2 SQL data types. Currently CLI is not able to determine if a column is an identity column, therefore SQL_FALSE is always returned. This limitation does not fully conform to the ODBC specifications. Future versions of CLI for UNIX and Windows servers will provide auto-unique support.
SQL_DESC_BASE_COLUMN_NAME (DB2 CLI v5)	Character AttributePtr	The base column name for the set column. If a base column name does not exist (as in the case of columns that are expressions), then this variable contains an empty string. This information is returned from the SQL_DESC_BASE_COLUMN_NAME record field of the IRD, which is a read-only field.
SQL_DESC_BASE_TABLE_NAME (DB2 CLI v5)	Character AttributePtr	The name of the base table that contains the column. If the base table name cannot be defined or is not applicable, then this variable contains an empty string.
SQL_DESC_CASE_SENSITIVE (DB2 CLI v2)	Numeric AttributePtr	Indicates if the column data type is a case sensitive data type. Either SQL_TRUE or SQL_FALSE will be returned in *NumericAttributePtr* depending on the data type. Case sensitivity does not apply to graphic data types, SQL_FALSE is returned. SQL_FALSE is returned for non-character data types and for the XML data type.
SQL_DESC_CATALOG_NAME (DB2 CLI v2)	Character AttributePtr	An empty string is returned since CLI only supports two part naming for a table.

Table 21. SQLColAttribute arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_CONCISE_TYPE (DB2 CLI v5)	Numeric AttributePtr	The concise data type. For the datetime data types, this field returns the concise data type, for example, SQL_TYPE_TIME. This information is returned from the SQL_DESC_CONCISE_TYPE record field of the IRD.
SQL_DESC_COUNT (DB2 CLI v2)	Numeric AttributePtr	The number of columns in the result set is returned in *NumericAttributePtr*.
SQL_DESC_DISPLAY_SIZE (DB2 CLI v2)	Numeric AttributePtr	The maximum number of bytes needed to display the data in character form is returned in *NumericAttributePtr*. Refer to the data type display size table for the display size of each of the column types.
SQL_DESC_DISTINCT_TYPE (DB2 CLI v2)	Character AttributePtr	The user defined distinct type name of the column is returned in *CharacterAttributePtr*. If the column is a built-in SQL type and not a user defined distinct type, an empty string is returned. **Note:** This is an IBM defined extension to the list of descriptor attributes defined by ODBC.
SQL_DESC_FIXED_PREC_SCALE (DB2 CLI v2)	Numeric AttributePtr	SQL_TRUE if the column has a fixed precision and non-zero scale that are data-source-specific. SQL_FALSE if the column does not have a fixed precision and non-zero scale that are data-source-specific. SQL_FALSE is returned in *NumericAttributePtr* for all DB2 SQL data types.
SQL_DESC_LABEL (DB2 CLI v2)	Character AttributePtr	The column label is returned in *CharacterAttributePtr*. If the column does not have a label, the column name or the column expression is returned. If the column is unlabeled and unnamed, an empty string is returned.
SQL_DESC_LENGTH (DB2 CLI v2)	Numeric AttributePtr	A numeric value that is either the maximum or actual element (SQLCHAR or SQLWCHAR) length of a character string or binary data type. It is the maximum element length for a fixed-length data type, or the actual element length for a variable-length data type. Its value always excludes the null termination byte that ends the character string. This information is returned from the SQL_DESC_LENGTH record field of the IRD. This value is 0 for the XML data type.
SQL_DESC_LITERAL_PREFIX (DB2 CLI v5)	Character AttributePtr	This VARCHAR(128) record field contains the character or characters that CLI recognizes as a prefix for a literal of this data type. This field contains an empty string for a data type for which a literal prefix is not applicable.

Table 21. SQLColAttribute arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_LITERAL_SUFFIX (DB2 CLI v5)	Character AttributePtr	This VARCHAR(128) record field contains the character or characters that CLI recognizes as a suffix for a literal of this data type. This field contains an empty string for a data type for which a literal suffix is not applicable.
SQL_DESC_LOCAL_TYPE_NAME (DB2 CLI v5)	Character AttributePtr	This VARCHAR(128) record field contains any localized (native language) name for the data type that might be different from the regular name of the data type. If there is no localized name, then an empty string is returned. This field is for display purposes only. The character set of the string is locale-dependent and is typically the default character set of the server.
SQL_DESC_NAME (DB2 CLI v2)	Character AttributePtr	The name of the column *ColumnNumber* is returned in *CharacterAttributePtr*. If the column is an expression, then the column number is returned. In either case, SQL_DESC_UNNAMED is set to SQL_NAMED. If there is no column name or a column alias, an empty string is returned and SQL_DESC_UNNAMED is set to SQL_UNNAMED. This information is returned from the SQL_DESC_NAME record field of the IRD.
SQL_DESC_NULLABLE (DB2 CLI v2)	Numeric AttributePtr	If the column identified by *ColumnNumber* can contain nulls, then SQL_NULLABLE is returned in *NumericAttributePtr*. If the column is constrained not to accept nulls, then SQL_NO_NULLS is returned in *NumericAttributePtr*. This information is returned from the SQL_DESC_NULLABLE record field of the IRD.
SQL_DESC_NUM_PREX_RADIX (DB2 CLI v5)	Numeric AttributePtr	• If the data type in the SQL_DESC_TYPE field is an approximate data type, this SQLINTEGER field contains a value of 2 because the SQL_DESC_PRECISION field contains the number of bits. • If the data type in the SQL_DESC_TYPE field is an exact numeric data type, this field contains a value of 10 because the SQL_DESC_PRECISION field contains the number of decimal digits.

Table 21. SQLColAttribute arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_OCTET_LENGTH (DB2 CLI v2)	Numeric AttributePtr	The number of bytes of data associated with the column is returned in *NumericAttributePtr*. This is the length in bytes of data transferred on the fetch or SQLGetData() for this column if SQL_C_DEFAULT is specified as the C data type. Refer to data type length table for the length of each of the SQL data types. If the column identified in *ColumnNumber* is a fixed length character or binary string, (for example, SQL_CHAR or SQL_BINARY) the actual length is returned. If the column identified in *ColumnNumber* is a variable length character or binary string, (for example, SQL_VARCHAR or SQL_BLOB) the maximum length is returned. If the column identified in *ColumnNumber* is of type SQL_XML, 0 is returned.
SQL_DESC_PRECISION (DB2 CLI v2)	Numeric AttributePtr	The precision in units of digits is returned in *NumericAttributePtr* if the column is SQL_DECIMAL, SQL_NUMERIC, SQL_DOUBLE, SQL_FLOAT, SQL_INTEGER, SQL_REAL or SQL_SMALLINT. If the column is a character SQL data type, then the precision returned in *NumericAttributePtr*, indicates the maximum number of SQLCHAR or SQLWCHAR elements the column can hold. If the column is a graphic SQL data type, then the precision returned in *NumericAttributePtr*, indicates the maximum number of double-byte elements the column can hold. If the column is the XML data type, the precision is 0. Refer to data type precision table for the precision of each of the SQL data types. This information is returned from the SQL_DESC_PRECISION record field of the IRD.
SQL_DESC_SCALE (DB2 CLI v2)	Numeric AttributePtr	The scale attribute of the column is returned. Refer to the data type scale table for the scale of each of the SQL data types. This information is returned from the SCALE record field of the IRD.
SQL_DESC_SCHEMA_NAME (DB2 CLI v2)	Character AttributePtr	The schema of the table that contains the column is returned in *CharacterAttributePtr*. The name of the schema that contains the table is returned. If the schema name is of less than 8 characters, then spaces are appended as extra characters.

Table 21. SQLColAttribute arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_SEARCHABLE (DB2 CLI v2)	Numeric AttributePtr	Indicates if the column data type is searchable: • SQL_PRED_NONE (SQL_UNSEARCHABLE in DB2 CLI v2) if the column cannot be used in a WHERE clause. • SQL_PRED_CHAR (SQL_LIKE_ONLY in DB2 CLI v2) if the column can be used in a WHERE clause only with the LIKE predicate. • SQL_PRED_BASIC (SQL_ALL_EXCEPT_LIKE in DB2 CLI v2) if the column can be used in a WHERE clause with all comparison operators except LIKE. • SQL_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator.
SQL_DESC_TABLE_NAME (DB2 CLI v2)	Character AttributePtr	The name of the table that contains the column is returned. If the table name cannot be defined or is not applicable, then this variable contains an empty string.
SQL_DESC_TYPE (DB2 CLI v2)	Numeric AttributePtr	The SQL data type of the column identified in *ColumnNumber* is returned in *NumericAttributePtr*. The possible values returned are listed in table of symbolic and default data types for CLI. When *ColumnNumber* is equal to 0, SQL_BINARY is returned for variable-length bookmarks, and SQL_INTEGER is returned for fixed-length bookmarks. For the datetime data types, this field returns the verbose data type, for example, SQL_DATETIME. This information is returned from the SQL_DESC_TYPE record field of the IRD.
SQL_DESC_TYPE_NAME (DB2 CLI v2)	Character AttributePtr	The type of the column (as entered in an SQL statement) is returned in *CharacterAttributePtr*. For information about each data type, refer to the list of symbolic and default data types for CLI.
SQL_DESC_UNNAMED (DB2 CLI v5)	Numeric AttributePtr	SQL_NAMED or SQL_UNNAMED. If the SQL_DESC_NAME field of the IRD contains a column alias, or a column name, SQL_NAMED is returned. If there is no column name or a column alias, SQL_UNNAMED is returned. This information is returned from the SQL_DESC_UNNAMED record field of the IRD.
SQL_DESC_UNSIGNED (DB2 CLI v2)	Numeric AttributePtr	Indicates if the column data type is an unsigned type or not. SQL_TRUE is returned in *NumericAttributePtr* for all non-numeric data types, SQL_FALSE is returned for all numeric data types.

Table 21. SQLColAttribute arguments (continued)

FieldIdentifier	**Information returned in**	**Description**
SQL_DESC_UPDATABLE (DB2 CLI v2)	Numeric AttributePtr	Indicates if the column data type is an updatable data type: • SQL_ATTR_READONLY is returned if the result set column is read-only. • SQL_ATTR_WRITE is returned if the result set column is read-write. • SQL_ATTR_READWRITE_UNKNOWN is returned if it is not known whether the result set column is updatable or not.

This function is an extensible alternative to SQLDescribeCol(). SQLDescribeCol() returns a fixed set of descriptor information based on ANSI-89 SQL. SQLColAttribute() allows access to the more extensive set of descriptor information available in ANSI SQL-92 and DBMS vendor extensions.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 22. SQLColAttribute SQLSTATEs

SQLSTATE	**Description**	**Explanation**
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The buffer **CharacterAttributePtr* was not large enough to return the entire string value, so the string was truncated. The length of the untruncated string value is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)
07005	The statement did not return a result set.	The statement associated with the *StatementHandle* did not return a result set. There were no columns to describe.
07009	Invalid descriptor index.	The value specified for *ColumnNumber* was equal to 0, and the SQL_ATTR_USE_BOOKMARKS statement attribute was SQL_UB_OFF. The value specified for the argument *ColumnNumber* was greater than the number of columns in the result set.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.

Table 22. SQLColAttribute SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The function was called before calling SQLPrepare() or SQLExecDirect() for the *StatementHandle.* An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. SQLExecute() or SQLExecDirect() was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* was less than 0.
HY091	Invalid descriptor field identifier.	The value specified for the argument *FieldIdentifier* was not one of the defined values, and was not an implementation-defined value.
HYC00	Driver not capable.	The value specified for the argument *FieldIdentifier* was not supported by CLI.

SQLColAttribute() can return any SQLSTATE that can be returned by SQLPrepare() or SQLExecute() when called after SQLPrepare() and before SQLExecute() depending on when the data source evaluates the SQL statement associated with the *StatementHandle*.

For performance reasons, an application should not call SQLColAttribute() before executing a statement.

Restrictions

None.

Example

```
/* get display size for column */
cliRC = SQLColAttribute(hstmt,
                        (SQLSMALLINT)(i + 1),
                        SQL_DESC_DISPLAY_SIZE,
                        NULL,
                        0,
                        NULL,
                        &colDataDisplaySize)
```

SQLColAttributes function (CLI) - Get column attributes

In ODBC 3.0, SQLColAttributes() has been deprecated and replaced with SQLColAttribute().

Although this version of CLI continues to support SQLColAttributes(), use SQLColAttribute() in your CLI programs so that they conform to the latest standards.

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLColAttributesW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Migrating to the new function

The statement:

```
SQLColAttributes (hstmt, colNum, SQL_DESC_COUNT, NULL, len,
                  NULL, &numCols);
```

for example, would be rewritten using the new function as:

```
SQLColAttribute (hstmt, colNum, SQL_DESC_COUNT, NULL, len,
                 NULL, &numCols);
```

SQLColumnPrivileges function (CLI) - Get privileges associated with the columns of a table

Returns a list of columns and associated privileges for the specified table.

The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated from a query.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

`SQLColumnPrivileges()` returns a list of columns and associated privileges for the specified table. The information is returned in an SQL result set, which you can retrieve by using the same functions that you use to process a result set that is generated from a query.

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is `SQLColumnPrivilegesW()`. See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN SQLColumnPrivileges(
             SQLHSTMT          StatementHandle,   /* hstmt */
             SQLCHAR           *CatalogName,      /* szCatalogName */
             SQLSMALLINT       NameLength1,       /* cbCatalogName */
             SQLCHAR           *SchemaName,       /* szSchemaName */
             SQLSMALLINT       NameLength2,       /* cbSchemaName */
             SQLCHAR           *TableName         /* szTableName */
             SQLSMALLINT       NameLength3,       /* cbTableName */
             SQLCHAR           *ColumnName,       /* szColumnName */
             SQLSMALLINT       NameLength4);      /* cbColumnName */
```

Function arguments

Table 23. SQLColumnPrivileges arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	The statement handle.

Table 23. SQLColumnPrivileges arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*CatalogName*	Input	The catalog qualifier of a 3-part table name. If the target DBMS does not support 3-part naming, and *CatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS is returned. Otherwise, this is a valid filter for DBMSs that support 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.
SQLCHAR *	*SchemaName*	Input	The schema qualifier of the table name.
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*TableName*	Input	The table name.
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *TableName*, or SQL_NTS if *TableName* is null-terminated.
SQLCHAR *	*ColumnName*	Input	A buffer that might contain a *pattern value* to qualify the result set by column name.
SQLSMALLINT	*NameLength4*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *ColumnName*, or SQL_NTS if *ColumnName* is null-terminated.

Usage

The results are returned as a standard result set that contains the columns listed in Columns Returned by SQLColumnPrivileges. The result set is ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, COLUMN_NAME, and PRIVILEGE. If multiple privileges are associated with any given column, each privilege is returned as a separate row. A typical application might want to call this function after a call to `SQLColumns()` to determine column privilege information. The application should use the character strings that are returned in the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, COLUMN_NAME columns of the `SQLColumns()` result set as input arguments to this function.

Because calls to `SQLColumnPrivileges()`, in many cases, map to a complex and thus expensive query against the system catalog, you should use the calls sparingly, and save the results rather than repeating the calls.

The *ColumnName* input argument accepts a search pattern, however, all other input arguments do not.

Sometimes, an application calls the function and no attempt is made to restrict the result set that is returned. In order to help reduce the long retrieval times, you can specify the configuration keyword SchemaList in the CLI initialization file to help restrict the result set when the application has supplied a null pointer for SchemaName. If the application specifies a SchemaName string, the SchemaList

keyword is still used to restrict the output. Therefore, if the schema name that is supplied is not in the SchemaList string, the result is an empty result set.

You can specify *ALL or *USRLIBL as values in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. If you specify *ALL, CLI searches on all existing schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. For IBM DB2 for IBM i servers, if you specify *USRLIBL, CLI searches on the current libraries of the server job. For other DB2 servers, *USRLIBL does not have a special meaning and CLI searches using *USRLIBL as a pattern. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL or *USRLIBL.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns returned by SQLColumnPrivileges

Column 1 TABLE_CAT (VARCHAR(128) Data type)
Name of the catalog. The value is NULL if this table does not have catalogs.

Column 2 TABLE_SCHEM (VARCHAR(128))
Name of the schema containing TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128) not NULL)
Name of the table or view.

Column 4 COLUMN_NAME (VARCHAR(128) not NULL)
Name of the column of the specified table or view.

Column 5 GRANTOR (VARCHAR(128))
Authorization ID of the user who granted the privilege.

Column 6 GRANTEE (VARCHAR(128))
Authorization ID of the user to whom the privilege is granted.

Column 7 PRIVILEGE (VARCHAR(128))
The column privilege. This can be:
- INSERT
- REFERENCES
- SELECT
- UPDATE

Note: Some IBM RDBMSs do not offer column level privileges at the column level. DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, and DB2 Server for VM and VSE support the UPDATE column privilege; there is one row in this result set for each updateable column. For all other privileges for DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, and DB2 Server for VM and VSE, and for all privileges for other IBM RDBMSs, if a privilege has been granted at the table level, a row is present in this result set.

Column 8 IS_GRANTABLE (VARCHAR(3) Data type)
Indicates whether the grantee is permitted to grant the privilege to other users.

Either “YES” or “NO”.

Note: The column names that are used by CLI follow the X/Open CLI CAE specification style. The column types, contents, and order are identical to those defined for the `SQLColumnPrivileges()` result set in ODBC.

If there is more than one privilege associated with a column, each privilege is returned as a separate row in the result set.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 24. SQLColumnPrivileges SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40001	Serialization failure	The transaction was rolled back due to a resource deadlock with another transaction.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY009	Invalid argument value.	*TableName* is NULL.
HY010	Function sequence error	An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. `SQLExecute()`, `SQLExecDirect()`, or `SQLSetPos()` was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

Example

```
cliRC = SQLColumnPrivileges(hstmt,
                            NULL,
                            0,
                            tbSchema,
                            SQL_NTS,
                            tbName,
                            SQL_NTS,
                            colNamePattern,
                            SQL_NTS);
```

SQLColumns function (CLI) - Get column information for a table

The SQLColumns() function returns a list of columns in the specified tables. The information is returned in an SQL result set, which you can retrieve by using the same functions that you use to fetch a result set that is generated by a query.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

Unicode Equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is SQLColumnsW(). For details about ANSI to Unicode function mappings, see "Unicode functions (CLI)" on page 5.

Syntax

```
SQLRETURN   SQLColumns      (
               SQLHSTMT         StatementHandle,   /* hstmt */
               SQLCHAR          *CatalogName,      /* szCatalogName */
               SQLSMALLINT      NameLength1,       /* cbCatalogName */
               SQLCHAR          *SchemaName,       /* szSchemaName */
               SQLSMALLINT      NameLength2,       /* cbSchemaName */
               SQLCHAR          *TableName,        /* szTableName */
               SQLSMALLINT      NameLength3,       /* cbTableName */
               SQLCHAR          *ColumnName,       /* szColumnName */
               SQLSMALLINT      NameLength4);      /* cbColumnName */
```

Function arguments

Table 25. SQLColumns arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	The statement handle.
SQLCHAR *	*CatalogName*	Input	A catalog qualifier of a 3-part table name. If the target DBMS does not support 3-part naming, and *CatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS will be returned. Otherwise, this is a valid filter for DBMSs that supports 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that is required to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.
SQLCHAR *	*SchemaName*	Input	A buffer that might contain a *pattern value* to qualify the result set by schema name.

Table 25. SQLColumns arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*TableName*	Input	A buffer that might contain a *pattern value* to qualify the result set by table name.
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *TableName*, or SQL_NTS if *TableName* is null-terminated.
SQLCHAR *	*ColumnName*	Input	A buffer that might contain a *pattern value* to qualify the result set by column name.
SQLSMALLINT	*NameLength4*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *ColumnName*, or SQL_NTS if *ColumnName* is null-terminated.

Usage

Use this function to retrieve information about the columns of either a table or a set of tables. An application can call this function after a call to `SQLTables()` to determine the columns of a table. The application must use the character strings that are returned in the TABLE_SCHEMA and TABLE_NAME columns of the `SQLTables()` result set as input to this function.

The `SQLColumns()` function returns a standard result set that is ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and ORDINAL_POSITION. Columns returned by SQLColumns lists the columns that are in the result set.

The *SchemaName*, *TableName*, and *ColumnName* input arguments accept search patterns.

Sometimes, an application calls the function and no attempt is made to restrict the result set that is returned. For some data sources that contain a large quantity of tables, views, and aliases for example, this scenario maps to an extremely large result set and very long retrieval times. In order to help reduce the long retrieval times, you can specify the configuration keyword **SchemaList** in the CLI initialization file to help restrict the result set when the application has supplied a null pointer for the **SchemaName**. If the application specifies a **SchemaName** string, the **SchemaList** keyword is still used to restrict the output. Therefore, if the schema name supplied is not in the **SchemaList** string, the result will be an empty result set.

This function does not return information about the columns of a result set. Instead, you should use `SQLDescribeCol()` or `SQLColAttribute()` function.

If the SQL_ATTR_LONGDATA_COMPAT attribute is set to SQL_LD_COMPAT_YES via either a call to `SQLSetConnectAttr()` or by setting the LONGDATACOMPAT keyword in the CLI initialization file, then the LOB data types are reported as SQL_LONGVARCHAR, SQL_LONGVARBINARY or SQL_LONGVARGRAPHIC.

In many cases, calls to the SQLColumns() function map to a complex and thus expensive query against the system catalog, so you should use the calls sparingly, and save the results rather than repeating calls.

Call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns that are supported by the connected DBMS.

You can specify *ALL or *USRLIBL as values in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. If you specify *ALL, CLI searches on all existing schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. For IBM DB2 for IBM i servers, if you specify *USRLIBL, CLI searches on the current libraries of the server job. For other DB2 servers, *USRLIBL does not have a special meaning and CLI searches using *USRLIBL as a pattern. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL or *USRLIBL.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns returned by SQLColumns

Column 1 TABLE_CAT (VARCHAR(128))
The name of the catalog. The value is NULL if this table does not have catalogs.

Column 2 TABLE_SCHEM (VARCHAR(128))
The name of the schema containing TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128) not NULL)
The name of the table, view, alias, or synonym.

Column 4 COLUMN_NAME (VARCHAR(128) not NULL)
The column identifier. The name of the column of the specified table, view, alias, or synonym.

Column 5 DATA_TYPE (SMALLINT not NULL)
The SQL data type of the column that is identified by COLUMN_NAME. The DATA_TYPE is one of the values in the Symbolic SQL Data Type column in the table of symbolic and default data types for CLI.

Column 6 TYPE_NAME (VARCHAR(128) not NULL)
A character string that represents the name of the data type that corresponds to DATA_TYPE.

Column 7 COLUMN_SIZE (INTEGER)
If the DATA_TYPE column value denotes a character or binary string, this column contains the maximum length in SQLCHAR or SQLWCHAR elements for the column.

For date, time, and timestamp data types, the COLUMN_SIZE is the total number of SQLCHAR or SQLWCHAR elements that are required to display the value when converted to character data type.

For numeric data types, the COLUMN_SIZE is either the total number of digits or the total number of bits that are allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.

For the XML data type, the length of zero is returned.

See the table of data type precision.

Column 8 BUFFER_LENGTH (INTEGER)
The maximum number of bytes for the associated C buffer to store data from this column if SQL_C_DEFAULT is specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` calls. This length does not include any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign.

See the table of data type lengths.

Column 9 DECIMAL_DIGITS (SMALLINT)
The scale of the column. NULL is returned for data types where scale is not applicable.

See the table of data type scale.

Column 10 NUM_PREC_RADIX (SMALLINT)
Either 10, 2, or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, and the COLUMN_SIZE column contains the number of bits that are allowed in the column.

If DATA_TYPE is an exact numeric data type, this column contains the value 10, and the COLUMN_SIZE contains the number of decimal digits that are allowed for the column.

For numeric data types, the DBMS can return a NUM_PREC_RADIX of 10 or 2.

NULL is returned for data types where the radix is not applicable.

Column 11 NULLABLE (SMALLINT not NULL)
SQL_NO_NULLS if the column does not accept NULL values.

SQL_NULLABLE if the column accepts NULL values.

Column 12 REMARKS (VARCHAR(254))
Might contain descriptive information about the column. It is possible that no information is returned in this column. For more details, see Optimize SQL columns keyword and attribute.

Column 13 COLUMN_DEF (VARCHAR(254))
The default value of the column. If the default value is a numeric literal, this column contains the character representation of the numeric literal with no enclosing single quotation marks. If the default value is a character string, this column is that string that is enclosed in single quotation marks. If the default value is a *pseudo-literal*, such as for DATE, TIME, and TIMESTAMP columns, this column contains the keyword of the pseudo-literal (for example. CURRENT DATE) with no enclosing quotation marks.

If NULL is specified as the default value, this column returns the word NULL, not enclosed in quotation marks. If the default value cannot be represented without truncation, this column contains TRUNCATED with no enclosing single quotation marks. If no default value is specified, this column is NULL.

It is possible that no information is returned in this column. For more details, see Optimize SQL columns keyword and attribute.

Column 14 SQL_DATA_TYPE (SMALLINT not NULL)
The SQL data type, as it is displayed in the SQL_DESC_TYPE record field in the IRD. This column is the same as the DATA_TYPE column in Columns returned by SQLColumns for the date, time, and timestamp data types.

Column 15 SQL_DATETIME_SUB (SMALLINT)
The subtype code for datetime data types:
- SQL_CODE_DATE
- SQL_CODE_TIME
- SQL_CODE_TIMESTAMP

For all other data types this column returns NULL.

Column 16 CHAR_OCTET_LENGTH (INTEGER)
For single byte character sets, this is the same as COLUMN_SIZE. For the XML type, zero is returned. For all other data types, NULL is returned.

Column 17 ORDINAL_POSITION (INTEGER not NULL)
The ordinal position of the column in the table. The first column in the table is number 1.

Column 18 IS_NULLABLE (VARCHAR(254))
Contains the string 'NO' if the column is known to be not nullable, and 'YES' if the column is nullable.

Note: This result set is identical to the X/Open CLI `Columns()` result set specification, which is an extended version of the `SQLColumns()` result set that is specified in ODBC V2. The ODBC `SQLColumns()` result set includes every column in the same position.

Optimize SQL columns keyword and attribute

It is possible to set up the CLI/ODBC Driver to optimize calls to the `SQLColumns()` function by using either:
- OPTIMIZESQLCOLUMNS CLI/ODBC configuration keyword
- SQL_ATTR_OPTIMIZESQLCOLUMNS connection attribute of SQLSetConnectAttr()

If either of these values are set, the information that is contained in the succeeding columns is not returned:
- Column 12 REMARKS
- Column 13 COLUMN_DEF

Return codes
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_STILL_EXECUTING
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

Diagnostics

Table 26. SQLColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. You can set the timeout period by using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restriction

The SQLColumns() function does not support returning data from an alias of an alias. When called against an alias of an alias, the SQLColumns() function returns an empty result set.

Example

```
/* get column information for a table */
cliRC = SQLColumns(hstmt,
                   NULL,
                   0,
                   tbSchemaPattern,
                   SQL_NTS,
                   tbNamePattern,
                   SQL_NTS,
                   colNamePattern,
                   SQL_NTS);
```

SQLConnect function (CLI) - Connect to a data source

Establishes a connection or a trusted connection to the target database.

The application must supply a target SQL database, and optionally an authorization-name and an authentication-string.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

A connection must be established before allocating a statement handle using `SQLAllocHandle()`.

Unicode Equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLConnectW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLConnect       (
               SQLHDBC           ConnectionHandle,       /* hdbc */
               SQLCHAR           *ServerName,             /* szDSN */
               SQLSMALLINT       ServerNameLength,       /* cbDSN */
               SQLCHAR           *UserName,               /* szUID */
               SQLSMALLINT       UserNameLength,         /* cbUID */
               SQLCHAR           *Authentication,         /* szAuthStr */
               SQLSMALLINT       AuthenticationLength);  /* cbAuthStr */
```

Function arguments

Table 27. SQLConnect arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection handle
SQLCHAR *	*ServerName*	input	Data Source: The name or alias-name of the database.
SQLSMALLINT	*ServerNameLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the *ServerName* argument.
SQLCHAR *	*UserName*	input	Authorization-name (user identifier)
SQLSMALLINT	*UserNameLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the *UserName* argument.
SQLCHAR *	*Authentication*	input	Authentication-string (password)
SQLSMALLINT	*AuthenticationLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the *Authentication* argument.

Usage

The target database (also known as *data source*) for IBM RDBMSs is the database-alias. The application can obtain a list of databases available to connect to by calling `SQLDataSources()`.

The input length arguments to SQLConnect() (*ServerNameLength*, *UserNameLength*, *AuthenticationLength*) can be set to the actual length of their associated data in elements (SQLCHAR or SQLWCHAR), not including any null-terminating character, or to SQL_NTS to indicate that the associated data is null-terminated.

The *ServerName* and *UserName* argument values must not contain any blanks.

Stored procedures written using CLI must make a *null* SQLConnect() call. A null SQLConnect() is where the *ServerName*, *UserName*, and *Authentication* argument pointers are all set to NULL and their length arguments are all set to 0. A null SQLConnect() still requires `SQLAllocHandle()` to be called first, but does not require that `SQLEndTran()` be called before `SQLDisconnect()`.

To create a trusted connection, specify the connection attribute SQL_ATTR_USE_TRUSTED_CONTEXT before calling `SQLConnect()`. If the database server accepts the connection as trusted the connection is treated as a trusted connection. Otherwise the connection is a regular connection and a warning is returned.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 28. SQLConnect SQLSTATEs

SQLSTATE	Description	Explanation
01679	Unable to establish a trusted connection.	CLI requested a trusted connection but the trust attributes of the connection do not match any trusted context object on the database server. The connection is allowed but it is a regular connection, not a trusted connection.
08001	Unable to connect to data source.	CLI was unable to establish a connection with the data source (server). The connection request was rejected because an existing connection established via embedded SQL already exists.
08002	Connection in use.	The specified *ConnectionHandle* has already been used to establish a connection with a data source and the connection is still open.
08004	The application server rejected establishment of the connection.	The data source (server) rejected the establishment of the connection.
28000	Invalid authorization specification.	The value specified for the argument *UserName* or the value specified for the argument *Authentication* violated restrictions defined by the data source.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Table 28. SQLConnect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY090	Invalid string or buffer length.	The value specified for argument *ServerNameLength* was less than 0, but not equal to SQL_NTS and the argument *ServerName* was not a null pointer. The value specified for argument *UserNameLength* was less than 0, but not equal to SQL_NTS and the argument *UserName* was not a null pointer. The value specified for argument *AuthenticationLength* was less than 0, but not equal to SQL_NTS and the argument *Authentication* was not a null pointer.
HY501	Invalid data source name.	An invalid data source name was specified in argument *ServerName*.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

The implicit connection (or default database) option for IBM RDBMSs is not supported. SQLConnect() must be called before any SQL statements can be executed.

Example

```
/* connect to the database */
cliRC = SQLConnect(hdbc,
                   (SQLCHAR *)db1Alias,
                   SQL_NTS,
                   (SQLCHAR *)user,
                   SQL_NTS,
                   (SQLCHAR *)pswd,
                   SQL_NTS);
```

SQLCopyDesc function (CLI) - Copy descriptor information between handles

Copies descriptor information from one descriptor handle to another.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Syntax

```
SQLRETURN   SQLCopyDesc (
               SQLHDESC       SourceDescHandle,   /* hDescSource */
               SQLHDESC       TargetDescHandle);  /* hDescTarget */
```

Function arguments

Table 29. SQLCopyDesc arguments

Data type	Argument	Use	Description
SQLHDESC	*SourceDescHandle*	input	Source descriptor handle.
SQLHDESC	*TargetDescHandle*	input	Target descriptor handle. *TargetDescHandle* can be a handle to an application descriptor or an IPD. SQLCopyDesc() will return SQLSTATE HY016 (Cannot modify an implementation descriptor) if *TargetDescHandle* is a handle to an IRD.

Usage

A call to SQLCopyDesc() copies the fields of the source descriptor handle to the target descriptor handle. Fields can only be copied to an application descriptor or an IPD, but not to an IRD. Fields can be copied from either an application or an implementation descriptor.

All fields of the descriptor, except SQL_DESC_ALLOC_TYPE (which specifies whether the descriptor handle was automatically or explicitly allocated), are copied, whether or not the field is defined for the destination descriptor. Copied fields overwrite the existing fields in the *TargetDescHandle*.

All descriptor fields are copied, even if *SourceDescHandle* and *TargetDescHandle* are on two different connections or environments.

The call to SQLCopyDesc() is immediately aborted if an error occurs.

When the SQL_DESC_DATA_PTR field is copied, a consistency check is performed. If the consistency check fails, SQLSTATE HY021 (Inconsistent descriptor information.) is returned and the call to SQLCopyDesc() is immediately aborted.

Note: Descriptor handles can be copied across connections or environments. An application may, however, be able to associate an explicitly allocated descriptor handle with a *StatementHandle*, rather than calling SQLCopyDesc() to copy fields from one descriptor to another. An explicitly allocated descriptor can be associated with another *StatementHandle* on the same *ConnectionHandle* by setting the SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC statement attribute to the handle of the explicitly allocated descriptor. When this is done, SQLCopyDesc() does not have to be called to copy descriptor field values from one descriptor to another.

A descriptor handle cannot be associated with a *StatementHandle* on another *ConnectionHandle*, however; to use the same descriptor field values on *StatementHandle* on different *ConnectionHandle*, SQLCopyDesc() has to be called.

Copying rows between tables

An ARD on one statement handle can serve as the APD on another statement handle. This allows an application to copy rows between tables without copying data at the application level. To do this, an application calls SQLCopyDesc() to copy the fields of an ARD that describes a fetched row of a table, to the APD for a parameter in an INSERT statement on another statement handle. The SQL_ACTIVE_STATEMENTS *InfoType* returned by the driver for a call to

SQLGetInfo() must be greater than 1 for this operation to succeed.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

When SQLCopyDesc() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec() with a `HandleType` of SQL_HANDLE_DESC and a `Handle` of *TargetDescHandle*. If an invalid *SourceDescHandle* was passed in the call, SQL_INVALID_HANDLE will be returned, but no SQLSTATE will be returned.

When an error is returned, the call to SQLCopyDesc() is immediately aborted, and the contents of the fields in the *TargetDescHandle* descriptor are undefined.

Table 30. SQLCopyDesc SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication link failure.	The communication link between CLI and the data source to which it was trying to connect failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY007	Associated statement is not prepared.	*SourceDescHandle* was associated with an IRD, and the associated statement handle was not in the prepared or executed state.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY016	Cannot modify an implementation row descriptor.	*TargetDescHandle* was associated with an IRD.
HY021	Inconsistent descriptor information.	The descriptor information checked during a consistency check was not consistent.
HY092	Option type out of range.	The call to SQLCopyDesc() prompted a call to SQLSetDescField(), but **ValuePtr* was not valid for the *FieldIdentifier* argument on *TargetDescHandle*.

Restrictions

None.

Example

```
SQLHANDLE hIRD, hARD; /* descriptor handles */

/* ... */

/* copy descriptor information between handles */
rc = SQLCopyDesc(hIRD, hARD);
```

SQLCreateDb function (CLI) - Create a database

The SQLCreateDb() function creates a database by using the specified database name, code set, and mode.

Specification:

- **CLI** V9.7
- **ODBC**
- **ISO CLI**

An active connection to the server must exist before you issue the SQLCreateDb API.

Unicode equivalent: The corresponding Unicode function is the SQLCreateDbW() function. For information about ANSI to Unicode function mappings, refer to "Unicode functions (CLI)" on page 5.

Syntax

```
SQLRETURN   SQL_API_FN SQLCreateDb ( SQLHDBC              hDbc,
                                     SQLCHAR              *szDbName,
                                     SQLINTEGER           cbDbName,
                                     SQLCHAR              *szCodeSet,
                                     SQLINTEGER           cbCodeSet,
                                     SQLCHAR              *szMode,
                                     SQLINTEGER           cbMode);
```

Function arguments

Table 31. SQLCreateDb arguments

Data type	Argument	Use	Description
SQLHDBC	*hDbc*	input	Connection handle.
SQLCHAR *	*szDbName*	input	Name of the database that is to be created.
SQLINTEGER	*cbDbName*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of the function) that is needed to store the *szDbName* argument or to store SQL_NTS if the *szDbName* argument is null terminated.
SQLCHAR *	*szCodeSet*	input	Database code set information. **Note:** If the value of the *szCodeSet*argument is NULL, the database is created in the Unicode code page for DB2 data servers and in the UTF-8 code page for IDS data servers.

Table 31. SQLCreateDb arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER	*cbCodeSet*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of the function) that is needed to store the *szCodeSet* argument or to store SQL_NTS if *szCodeSet* argument is null terminated.
SQLCHAR *	*szMode*	input	Database logging mode. **Note:** This value is applicable only to IDS data servers.
SQLINTEGER	*cbMode*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of the function) that is needed to store the *szMode* argument or to store SQL_NTS if *szMode* argument is null terminated.

Usage

When creating a DB2 database, CLI application must first connect to the server instance by specifying the ATTACH keyword. The valid APIs, after connecting to the server instance using ATTACH keyword are SQLCreateDb(), SQLDropDb(), and SQLDisconnect(). Before performing other CLI operations on the new database, you must disconnect from the server instance and then connect to the new database.

Return codes

- SQL_SUCCESS
- SQL_ERROR

Diagnostics

Table 32. SQLCreateDb SQLSTATEs

SQLSTATE	Description	Explanation
08003	Connection is closed.	The connection that was specified for the *SQLCreateDb* argument was not open.
HY090	Invalid string or buffer length.	The *cbDbName*, *cbCodeSet*, and *cbMode* arguments have a maximum length of 128. If you specify an invalid value, CLI generates an error.

Restrictions

- A connection representing an instance attachment is required.
- The SQLCreateDb() function is not supported for DB2 for IBM i and DB2 for z/OS servers.

.

Examples

The following example creates DB2 databases on a local server:

```
sqldriverconnect 1 0 "attach=true" -3 50 SQL_DRIVER_NOPROMPT
sqlcreatedb 1 sample1 8 null 0 null 0
sqlcreatedb 1 sample2 8 null 0 null 0
```

The following example creates DB2 databases on a remote server:

```
sqldriverconnect 1 0  "attach=true;hostname=myhostname;port=9999;
 uid=myuid;pwd=mypwd;protocol=tcpip" -3 50 SQL_DRIVER_NOPROMPT
sqlcreatedb 1 sample1 8 null 0 null 0
sqlcreatedb 1 sample2 8 null 0 null 0
```

Version information

Last update
This topic was last updated for IBM DB2 Version 9.7, Fix Pack 3.

IBM Data Server Client
Supported in IBM DB2 for Linux, UNIX, and Windows

SQLCreatePkg

`SQLCreatePkg()` invokes the bind utility, which prepares SQL statements stored in the bind file, and creates a package that is stored in the database.

Specification:

- **CLI** 9.5

Syntax

```
SQLRETURN   SQLCreatePkg(
               SQLHDBC          hDbc,
               SQLCHAR          *szBindFileNameIn,
               SQLINTEGER       cbBindFileNameIn,
               SQLCHAR          *szBindOpts,
               SQLINTEGER       cbBindOpts)
```

Function arguments

Table 33. `SQLCreatePkg()` arguments

Data type	Argument	Use	Description
SQLHDBC	*hDbc*	input	Connection handle.
SQLCHAR*	*szBindFileNameIn*	input	Name of the file to bind, or the name of a file containing a list of bind file names.
SQLINTEGER	*cbBindFileNameIn*	input	Number of SQLCHAR elements needed to store *szBindFileNameIn*, or `SQL_NTS` if *szBindFileNameIn* is null-terminated.
SQLCHAR*	*szBindOpts*	input	List of bind options separated by semicolon.
SQLINTEGER	*cbBindOpts*	input	Number of SQLCHAR elements needed to store *szBindOpts*, or `SQL_NTS` if *szBindOpts* is null-terminated.

Usage

The argument *szBindFileNameIn* is a string containing the name of the bind file, or the name of a file containing a list of bind file names. The bind file names must contain the extension `.bnd`. You can specify a path for these files. Precede the name of a bind list file with the at sign (@). The following example is a fully qualified bind list file name:

```
/u/user1/sqllib/bnd/@all.lst
```

The bind list file should contain one or more bind file names, and must have the extension `.lst`. Precede all but the first bind file name with a plus symbol (+). The bind file names can be on one or more lines. For example, the bind list file `all.lst` might contain the following lines:

```
mybind1.bnd+mybind2.bnd+
mybind3.bnd+
mybind4.bnd
```

You can use path specifications on bind file names in the list file. If no path is specified, the database manager takes path information from the bind list file.

The following **BIND** command parameters can be specified with `SQLCreatePkg()`:

- **KEEPDYNAMIC**={YES | NO}
- **ISOLATION**={CS | NC | RR | RS | UR}
- **BLOCKING**={YES | NO | UNAMBIG}
- **ENCODING**={ASCII | EBCDIC | UNICODE | CCSID | *integer*} (DB2 for z/OS and OS/390® only)
- **REOPT**={NONE | ONCE | ALWAYS}
- **COLLECTION**={schema name

The **BIND** command parameters can be passed in as a string with name-value pairs separated by a semicolon. For example:

```
keepdynamic=yes; isolation=cs; blocking=no
```

Both options and values are case insensitive.

Example 1: Binding a file with **REOPT**=ONCE and **ENCODING**=CCSID

```
strcpy (bindFileName, "insertEmp.bnd");
cliRC = SQLCreatePkg(hdbc,
                     bindFileName,
                     -3, // SQL_NTS
                     "REOPT=ONCE; ENCODING=CCSID");
```

Example 2: Binding a list of files all with **KEEPDYNAMIC**=YES, **BLOCKING**=NO, and **ISOLATION**=RS

```
strcpy (bindFileName, "/u/user1/sqllib/bnd/@all.lst");
cliRC = SQLCreatePkg(hdbc,
                     bindFileName,
                     strlen(bindFileName),
                     "KEEPDYNAMIC=YES; BLOCKING=NO; ISOLATION=RS");
```

Example 3: Binding a file with **COLLECTION**=SCHEMA NAME

```
strcpy (bindFileName, "insertEmp.bnd");
cliRC = SQLCreatePkg(hdbc,
                     bindFileName,
                     -3, // SQL_NTS
                     "REOPT=ONCE; ENCODING=CCSID;
            COLLECTION=NEWTON");
```

SQLDataSources function (CLI) - Get list of data sources

Returns a list of target databases available, one at a time.

A database must be cataloged to be available.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

SQLDataSources() is usually called before a connection is made, to determine the databases that are available to connect to.

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLDataSourcesW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLDataSources (
               SQLHENV           EnvironmentHandle,  /* henv */
               SQLUSMALLINT      Direction,          /* fDirection */
               SQLCHAR           *ServerName,        /* *szDSN */
               SQLSMALLINT       BufferLength1,      /* cbDSNMax */
               SQLSMALLINT       *NameLength1Ptr,    /* *pcbDSN */
               SQLCHAR           *Description,       /* *szDescription */
               SQLSMALLINT       BufferLength2,      /* cbDescriptionMax */
               SQLSMALLINT       *NameLength2Ptr);   /* *pcbDescription */
```

Function arguments

Table 34. SQLDataSources arguments

Data type	Argument	Use	Description
SQLHENV	*EnvironmentHandle*	input	Environment handle.
SQLUSMALLINT	*Direction*	input	Used by application to request the first data source name in the list or the next one in the list. *Direction* can take on only the following values: • SQL_FETCH_FIRST • SQL_FETCH_NEXT
SQLCHAR *	*ServerName*	output	Pointer to buffer in which to return the data source name.
SQLSMALLINT	*BufferLength1*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the *ServerName* buffer. This number must be less than or equal to SQL_MAX_DSN_LENGTH + 1.
SQLSMALLINT *	*NameLength1Ptr*	output	Pointer to a buffer in which to return the total number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function), excluding the null-termination character, available to return in **ServerName*. If the number of SQLCHAR or SQLWCHAR elements available to return is greater than or equal to *BufferLength1*, the data source name in **ServerName* is truncated to *BufferLength1* minus the length of a null-termination character.
SQLCHAR *	*Description*	output	Pointer to buffer where the description of the data source is returned. CLI will return the ***Comment*** field associated with the database catalogued to the DBMS.

Table 34. SQLDataSources arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*BufferLength2*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the *Description* buffer.
SQLSMALLINT *	*NameLength2Ptr*	output	Pointer to a buffer in which to return the total number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function), excluding the null-termination character, available to return in **Description*. If the number of SQLCHAR or SQLWCHAR elements available to return is greater than or equal to *BufferLength2*, the driver description in **Description* is truncated to *BufferLength2* minus the length of a null-termination character.

Usage

The application can call this function any time with *Direction* set to either SQL_FETCH_FIRST or SQL_FETCH_NEXT.

If SQL_FETCH_FIRST is specified, the first database in the list will always be returned.

If SQL_FETCH_NEXT is specified:
- Directly following a SQL_FETCH_FIRST call, the second database in the list is returned
- Before any other SQLDataSources() call, the first database in the list is returned
- When there are no more databases in the list, SQL_NO_DATA_FOUND is returned. If the function is called again, the first database is returned.
- Any other time, the next database in the list is returned.

In an ODBC environment, the ODBC Driver Manager will perform this function.

Since the IBM RDBMSs always returns the description of the data source blank padded to 30 bytes, CLI will do the same.

Return codes
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

Diagnostics

Table 35. SQLDataSources SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data source name returned in the argument *ServerName* was longer than the value specified in the argument *BufferLength1*. The argument *NameLength1Ptr* contains the length of the full data source name. (Function returns SQL_SUCCESS_WITH_INFO.) The data source name returned in the argument *Description* was longer than the value specified in the argument *BufferLength2*. The argument *NameLength2Ptr* contains the length of the full data source description. (Function returns SQL_SUCCESS_WITH_INFO.)
58004	Unexpected system failure.	Unrecoverable system error.
HY000	General error.	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by `SQLGetDiagRec()` in the *MessageText* argument describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for argument *BufferLength1* was less than 0. The value specified for argument *BufferLength2* was less than 0.
HY103	Direction option out of range.	The value specified for the argument *Direction* was not equal to SQL_FETCH_FIRST or SQL_FETCH_NEXT.

Authorization

None.

Example

```
/* get list of data sources */
cliRC = SQLDataSources(henv,
                       SQL_FETCH_FIRST,
                       dbAliasBuf,
                       SQL_MAX_DSN_LENGTH + 1,
                       &aliasLen,
                       dbCommentBuf,
                       255,
                       &commentLen);
```

SQLDescribeCol function (CLI) - Return a set of attributes for a column

Returns a set of commonly used descriptor information (column name, type, precision, scale, nullability) for the indicated column in the result set generated by a query.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

This information is also available in the fields of the IRD.

If the application needs only one attribute of the descriptor information, or needs an attribute not returned by SQLDescribeCol(), the SQLColAttribute() function can be used in place of SQLDescribeCol().

Either SQLPrepare() or SQLExecDirect() must be called before calling this function.

This function (or SQLColAttribute()) is usually called before a bind column function (SQLBindCol(), SQLBindFileToCol()) to determine the attributes of a column before binding it to an application variable.

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLDescribeColW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLDescribeCol (
               SQLHSTMT         StatementHandle,    /* hstmt */
               SQLUSMALLINT     ColumnNumber,       /* icol */
               SQLCHAR          *ColumnName,        /* szColName */
               SQLSMALLINT      BufferLength,       /* cbColNameMax */
               SQLSMALLINT      *NameLengthPtr,     /* pcbColName */
               SQLSMALLINT      *DataTypePtr,       /* pfSqlType */
               SQLULEN          *ColumnSizePtr,     /* pcbColDef */
               SQLSMALLINT      *DecimalDigitsPtr,  /* pibScale */
               SQLSMALLINT      *NullablePtr);      /* pfNullable */
```

Function arguments

Table 36. SQLDescribeCol arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLUSMALLINT	*ColumnNumber*	input	Column number to be described. Columns are numbered sequentially from left to right, starting at 1. This can also be set to 0 to describe the bookmark column.
SQLCHAR *	*ColumnName*	output	Pointer to column name buffer. This value is read from the SQL_DESC_NAME field of the IRD. This is set to NULL if the column name cannot be determined.
SQLSMALLINT	*BufferLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the * *ColumnName* buffer.

Table 36. SQLDescribeCol arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT *	*NameLengthPtr*	output	Pointer to a buffer in which to return the total number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function), excluding the null-termination character, available to return in * *ColumnName*. Truncation of column name (* *ColumnName*) to *BufferLength* - 1 SQLCHAR or SQLWCHAR elements occurs if *NameLengthPtr* is greater than or equal to *BufferLength*.
SQLSMALLINT *	*DataTypePtr*	output	Base SQL data type of column. To determine if there is a User Defined Type associated with the column, call SQLColAttribute() with *fDescType* set to SQL_COLUMN_DISTINCT_TYPE. Refer to the Symbolic SQL Data Type column of the symbolic and default data types table for the data types that are supported.
SQLULEN *	*ColumnSizePtr*	output	Precision of column as defined in the database. If *fSqlType* denotes a graphic or DBCLOB SQL data type, then this variable indicates the maximum number of double-byte *characters* the column can hold.
SQLSMALLINT *	*DecimalDigitsPtr*	output	Scale of column as defined in the database (only applies to SQL_DECIMAL, SQL_NUMERIC, SQL_TYPE_TIMESTAMP). Refer to the data type scale table for the scale of each of the SQL data types.
SQLSMALLINT *	*NullablePtr*	output	Indicates whether NULLS are allowed for this column • SQL_NO_NULLS • SQL_NULLABLE

Usage

Columns are identified by a number, are numbered sequentially from left to right, and can be described in any order.

- Column numbers start at 1 if bookmarks are not used (SQL_ATTR_USE_BOOKMARKS statement attribute set to SQL_UB_OFF).
- The *ColumnNumber* argument can be set to 0 to describe the bookmark column if bookmarks are used (the statement attribute is set to SQL_UB_ON).

If a null pointer is specified for any of the pointer arguments, CLI assumes that the information is not needed by the application and nothing is returned.

If the column is a User Defined Type, SQLDescribeCol() only returns the built-in type in *DataTypePtr*. Call SQLColAttribute() with *fDescType* set to SQL_COLUMN_DISTINCT_TYPE to obtain the User Defined Type.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

If SQLDescribeCol() returns either SQL_ERROR, or SQL_SUCCESS_WITH_INFO, one of the following SQLSTATEs can be obtained by calling the `SQLGetDiagRec()` or `SQLGetDiagField()` function.

Table 37. SQLDescribeCol SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The column name returned in the argument * *ColumnName* was longer than the value specified in the argument *BufferLength*. The argument * *NameLengthPtr* contains the length of the full column name. (Function returns SQL_SUCCESS_WITH_INFO.)
07005	The statement did not return a result set.	The statement associated with the *StatementHandle* did not return a result set. There were no columns to describe. (Call SQLNumResultCols() first to determine if there are any rows in the result set.)
07009	Invalid descriptor index	The value specified for *ColumnNumber* was equal to 0, and the SQL_ATTR_USE_BOOKMARKS statement attribute was SQL_UB_OFF. The value specified for the argument *ColumnNumber* was greater than the number of columns in the result set.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The function was called before calling SQLPrepare() or SQLExecDirect() for the *StatementHandle*. The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The length specified in argument *BufferLength* less than 1.
HYC00	Driver not capable.	The SQL data type of column *ColumnNumber* is not recognized by CLI.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

The following ODBC defined data types are not supported:
- SQL_BIT
- SQL_TINYINT

Example

```
/* return a set of attributes for a column */
cliRC = SQLDescribeCol(hstmt,
                       (SQLSMALLINT)(i + 1),
                       colName,
                       sizeof(colName),
                       &colNameLen,
                       &colType,
                       &colSize,
                       &colScale,
                       NULL);
```

SQLDescribeParam function (CLI) - Return description of a parameter marker

Returns the description of a parameter marker associated with a prepared SQL statement.

Specification:

- **CLI** 5.0
- **ODBC** 1.0
- **ISO CLI**

The description of a parameter marker is also available in the fields of the IPD.

If deferred prepared is enabled, and this is the first call to `SQLDescribeParam()`, `SQLNumResultCols()`, or `SQLDescribeCol()`, the call will force a PREPARE of the SQL statement to be flowed to the server.

Syntax

```
SQLRETURN   SQLDescribeParam (
               SQLHSTMT          StatementHandle,     /* hstmt */
               SQLUSMALLINT      ParameterNumber,     /* ipar */
               SQLSMALLINT       *DataTypePtr,        /* pfSqlType */
               SQLULEN           *ParameterSizePtr,   /* pcbParamDef */
               SQLSMALLINT       *DecimalDigitsPtr,   /* pibScale */
               SQLSMALLINT       *NullablePtr);       /* pfNullable */
```

Function arguments

Table 38. SQLDescribeParam arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLUSMALLINT	*ParameterNumber*	input	Parameter marker number ordered sequentially in increasing parameter order, starting at 1.

Table 38. SQLDescribeParam arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT *	*DataTypePtr*	output	Pointer to a buffer in which to return the SQL data type of the parameter. This value is read from the SQL_DESC_CONCISE_TYPE record field of the IPD. When ColumnNumber is equal to 0 (for a bookmark column), SQL_BINARY is returned in **DataTypePtr* for variable-length bookmarks.
SQLULEN *	*ParameterSizePtr*	output	Pointer to a buffer in which to return the size of the column or expression of the corresponding parameter marker as defined by the data source.
SQLSMALLINT *	*DecimalDigitsPtr*	output	Pointer to a buffer in which to return the number of decimal digits of the column or expression of the corresponding parameter as defined by the data source.
SQLSMALLINT *	*NullablePtr*	output	Pointer to a buffer in which to return a value that indicates whether the parameter allows NULL values. This value is read from the SQL_DESC_NULLABLE field of the IPD. The ODBC specification lists following returned values. However, the CLI driver only returns SQL_NULLABLE_UNKNOWN return value. • SQL_NO_NULLS: The parameter does not allow NULL values (this is the default value). • SQL_NULLABLE: The parameter allows NULL values. • SQL_NULLABLE_UNKNOWN: Cannot determine if the parameter allows NULL values. **Note:** The CLI driver returns SQL_NULLABLE_UNKNOWN.

Usage

Parameter markers are numbered in increasing order as they appear in the SQL statement, starting with 1.

SQLDescribeParam() does not return the type (input, input/output, or output) of a parameter in an SQL statement. Except in calls to stored procedures, all parameters in SQL statements are input parameters. To determine the type of each parameter in a call to a stored procedure, call `SQLProcedureColumns()`.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 39. SQLDescribeParam SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
07009	Invalid descriptor index.	The value specified for the argument *ParameterNumber* less than 1. The value specified for the argument *ParameterNumber* was greater than the number of parameters in the associated SQL statement. The parameter marker was part of a non-DML statement. The parameter marker was part of a SELECT list.
08S01	Communication link failure.	The communication link between CLI and the data source to which it was connected failed before the function completed processing.
21S01	Insert value list does not match column list.	The number of parameters in the INSERT statement did not match the number of columns in the table named in the statement.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The function was called before calling `SQLPrepare()` or `SQLExecDirect()` for the *StatementHandle*. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. `SQLExecute()` `SQLExecDirect()`, `SQLBulkOperations()`, or `SQLSetPos()` was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Unexpected memory handling error.	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HYC00	Driver not capable.	The schema function stored procedures are not accessible on the server. Install the schema function stored procedures on the server and ensure they are accessible.

Restrictions

None.

SQLDisconnect function (CLI) - Disconnect from a data source

Closes the connection associated with the database connection handle.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

SQLEndTran() must be called before calling SQLDisconnect() if an outstanding transaction exists on this connection.

After calling this function, either call SQLConnect() to connect to another database, or use SQLFreeHandle() to free the connection handle.

Syntax

```
SQLRETURN   SQLDisconnect (SQLHDBC       ConnectionHandle;) /* hdbc */
```

Function arguments

Table 40. SQLDisconnect arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection handle

Usage

If an application calls SQLDisconnect() before it has freed all the statement handles associated with the connection, CLI frees them after it successfully disconnects from the database.

If SQL_SUCCESS_WITH_INFO is returned, it implies that even though the disconnect from the database is successful, additional error or implementation specific information is available. For example, a problem was encountered on the clean up subsequent to the disconnect, or if there is no current connection because of an event that occurred independently of the application (such as communication failure).

After a successful SQLDisconnect() call, the application can re-use *ConnectionHandle* to make another SQLConnect() or SQLDriverConnect() request.

An application should not rely on SQLDisconnect() to close cursors (with both stored procedures and regular client applications). In both cases the cursor should be closed using SQLCloseCursor(), then the statement handle freed using SQLFreeHandle().

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 41. SQLDisconnect SQLSTATEs

SQLSTATE	Description	Explanation
01002	Disconnect error.	An error occurred during the disconnect. However, the disconnect succeeded. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	The connection specified in the argument *ConnectionHandle* was not open.
25000 **25**501	Invalid transaction state.	There was a transaction in process on the connection specified by the argument *ConnectionHandle*. The transaction remains active, and the connection cannot be disconnected. **Note:** This error does not apply to stored procedures written in CLI.
25501	Invalid transaction state.	There was a transaction in process on the connection specified by the argument *ConnectionHandle*. The transaction remains active, and the connection cannot be disconnected.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

```
SQLHANDLE hdbc; /* connection handle */

/* ... */

/* disconnect from the database */
  cliRC = SQLDisconnect(hdbc);
```

SQLDriverConnect function (CLI) - (Expanded) Connect to a data source

An alternative to SQLConnect(). Both functions establish a connection to the target database, but SQLDriverConnect() supports additional connection parameters and the ability to prompt the user for connection information.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

Use SQLDriverConnect() when the data source requires parameters other than the 3 input arguments supported by SQLConnect() (data source name, user ID and

password), or when you want to use CLI's graphical user interface to prompt the user for mandatory connection information.

Once a connection is established, the completed connection string is returned. Applications can store this string for future connection requests.

Syntax

Generic

```
SQLRETURN  SQLDriverConnect (
        SQLHDBC       ConnectionHandle,              /* hdbc */
        SQLHWND       WindowHandle,                  /* hwnd */
        SQLCHAR       *InConnectionString,           /* szConnStrIn */
        SQLSMALLINT   InConnectionStringLength,      /* cbConnStrIn */
        SQLCHAR       *OutConnectionString,          /* szConnStrOut */
        SQLSMALLINT   OutConnectionStringCapacity,   /* cbConnStrOutMax */
        SQLSMALLINT   *OutConnectionStringLengthPtr, /* pcbConnStrOut */
        SQLUSMALLINT  DriverCompletion);             /* fDriverCompletion */
```

Function arguments

Table 42. SQLDriverConnect arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection handle
SQLHWND	*WindowHandle*	input	Window handle. On Windows operating systems, this is the parent Windows handle. Currently the window handle is only supported on Windows. If a NULL is passed, then no dialog will be presented.
SQLCHAR *	*InConnectionString*	input	A full, partial or empty (null pointer) connection string (see following syntax and description).
SQLSMALLINT	*InConnectionStringLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store *InConnectionString*.
SQLSMALLINT *	*OutConnectionString*	output	Pointer to buffer for the completed connection string. If the connection was established successfully, this buffer will contain the completed connection string. Applications should allocate at least SQL_MAX_OPTION_STRING_LENGTH bytes for this buffer.
SQLSMALLINT	*OutConnectionString Capacity*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store *OutConnectionString*.
SQLSMALLINT *	*OutConnectionString LengthPtr*	output	Pointer to the number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function), excluding the null-termination character, available to return in the *OutConnectionString* buffer. If the value of **OutConnectionStringLengthPtr* is greater than or equal to *OutConnectionStringCapacity*, the completed connection string in *OutConnectionString* is truncated to *OutConnectionStringCapacity* - 1 SQLCHAR or SQLWCHAR elements.

Table 42. SQLDriverConnect arguments (continued)

Data type	Argument	Use	Description
SQLUSMALLINT	*DriverCompletion*	input	Indicates when CLI should prompt the user for more information. Possible values: • SQL_DRIVER_PROMPT • SQL_DRIVER_COMPLETE • SQL_DRIVER_COMPLETE_REQUIRED • SQL_DRIVER_NOPROMPT

Usage

InConnectionString Argument

A request connection string has the following syntax:

connection-string ::= attribute[;] | attribute; connection-string

attribute ::= attribute-keyword=attribute-value
| DRIVER=[{]attribute-value[}]

attribute-keyword ::= DSN | UID | PWD | NEWPWD
| driver-defined-attribute-keyword

attribute-value ::= character-string
driver-defined-attribute-keyword ::= identifier

where
- character-string has zero or more SQLCHAR or SQLWCHAR elements
- identifier has one or more SQLCHAR or SQLWCHAR elements
- attribute-keyword is case insensitive
- attribute-value may be case sensitive
- the value of the **DSN** keyword does not consist solely of blanks
- **NEWPWD** is used as part of a change password request. The application can either specify the new string to use, for example, `NEWPWD=anewpass;` or specify `NEWPWD=;` and rely on a dialog box generated by the CLI driver to prompt for the new password

Because of connection string and initialization file grammar, keywords and attribute values that contain the characters **[]{}(),;?*=!@** should be avoided. Because of the grammar in the system information, keywords and data source names cannot contain the backslash (\) character. For CLI Version 2, braces are required around the **DRIVER** keyword.

If any keywords are repeated in the browse request connection string, CLI uses the value associated with the first occurrence of the keyword. If the **DSN** and **DRIVER** keywords are included in the same browse request connection string, CLI uses whichever keyword appears first.

OutConnectionString Argument

The result connection string is a list of connection attributes. A connection attribute consists of an attribute keyword and a corresponding attribute value. The browse result connection string has the following syntax:

connection-string ::= attribute[;] | attribute; connection-string

attribute ::= [*]attribute-keyword=attribute-value
attribute-keyword ::= ODBC-attribute-keyword
| driver-defined-attribute-keyword

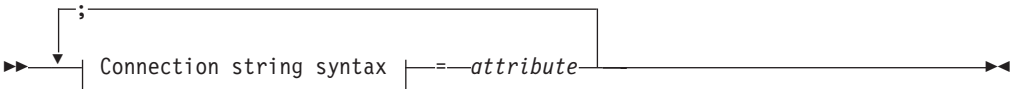
ODBC-attribute-keyword = {UID | PWD}:[localized-identifier]
driver-defined-attribute-keyword ::= identifier[:localized-identifier]

attribute-value ::= {attribute-value-list} | ?
(The braces are literal; they are returned by CLI.)
attribute-value-list ::= character-string [:localized-character string] | character-string [:localized-character string], attribute-value-list

where
- character-string and localized-character string have zero or more SQLCHAR or SQLWCHAR elements
- identifier and localized-identifier have one or more SQLCHAR or SQLWCHAR elements; attribute-keyword is case insensitive
- attribute-value may be case sensitive

Because of connection string and initialization file grammar, keywords, localized identifiers, and attribute values that contain the characters **[]{}(),;?*=!@** should be avoided. Because of the grammar in the system information, keywords and data source names cannot contain the backslash (\) character.

The connection string is used to pass one or more values needed to complete a connection. The contents of the connection string and the value of *DriverCompletion* will determine if CLI needs to establish a dialog with the user.



Connection string syntax

Attribute associated with each keyword are:

DSN Data source name. The name or alias-name of the database. Required if *DriverCompletion* is equal to SQL_DRIVER_NOPROMPT.

UID Authorization-name (user identifier).

PWD The password corresponding to the authorization name. If there is no password for the user ID, an empty value is specified (PWD=;).

NEWPWD
New password used as part of a change password request. The application can either specify the new string to use, for example, NEWPWD=anewpass; or specify NEWPWD=; and rely on a dialog box

generated by the CLI driver to prompt for the new password (set the *DriverCompletion* argument to anything other than SQL_DRIVER_NOPROMPT).

Any one of the CLI keywords can be specified on the connection string. If any keywords are repeated in the connection string, the value associated with the first occurrence of the keyword is used.

If any keywords exists in the CLI initialization file, the keywords and their values are used to augment the information passed to CLI in the connection string. If the information in the CLI initialization file contradicts information in the connection string, the values in connection string take precedence.

If the end user *Cancel*s a dialog box presented, SQL_NO_DATA_FOUND is returned.

The following values of *DriverCompletion* determines when a dialog will be opened:

SQL_DRIVER_PROMPT:
A dialog is always initiated. The information from the connection string and the CLI initialization file are used as initial values, to be supplemented by data input via the dialog box.

SQL_DRIVER_COMPLETE:
A dialog is only initiated if there is insufficient information in the connection string. The information from the connection string is used as initial values, to be supplemented by data entered via the dialog box.

SQL_DRIVER_COMPLETE_REQUIRED:
A dialog is only initiated if there is insufficient information in the connection string. The information from the connection string is used as initial values. Only mandatory information is requested. The user is prompted for required information only.

SQL_DRIVER_NOPROMPT:
The user is not prompted for any information. A connection is attempted with the information contained in the connection string. If there is not enough information, SQL_ERROR is returned.

Once a connection is established, the complete connection string is returned. Applications that need to set up multiple connections to the same database for a given user ID should store this output connection string. This string can then be used as the input connection string value on future SQLDriverConnect() calls.

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLDriverConnectW(). See “Unicode functions (CLI)” on page 5 for information about ANSI to Unicode function mappings.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

All of the diagnostics generated by SQLConnect() can be returned here as well. The following table shows the additional diagnostics that can be returned.

Table 43. SQLDriverConnect SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer *szConnstrOut* was not large enough to hold the entire connection string. The argument **OutConnectionStringLengthPtr* contains the actual length of the connection string available for return. (Function returns SQL_SUCCESS_WITH_INFO)
01S00	Invalid connection string attribute.	An invalid keyword or attribute value was specified in the input connection string, but the connection to the data source was successful anyway because one of the listed event has occurred: • The unrecognized keyword was ignored. • The invalid attribute value was ignored, the default value was used instead. (Function returns SQL_SUCCESS_WITH_INFO)
HY000	General error. Dialog Failed	The information specified in the connection string was insufficient for making a connect request, but the dialog was prohibited by setting *fCompletion* to SQL_DRIVER_NOPROMPT. The attempt to display the dialog failed.
HY090	Invalid string or buffer length.	The value specified for *InConnectionStringLength* was less than 0, but not equal to SQL_NTS. The value specified for *OutConnectionStringCapacity* was less than 0.
HY110	Invalid driver completion.	The value specified for the argument *fCompletion* was not equal to one of the valid values.

Restrictions

None.

Example

```
rc = SQLDriverConnect(hdbc,
                      (SQLHWND)sqlHWND,
                      InConnectionString,
                      InConnectionStringLength,
                      OutConnectionString,
                      OutConnectionStringCapacity,
                      StrLength2,
                      DriveCompletion);
```

SQLDropDb function (CLI) - Drop a database

The SQLDropDb() function drops the specified database.

Specification:

- **CLI** V9.7

Unicode Equivalent: The corresponding Unicode function is the SQLDropDbW() function. For information about ANSI to Unicode function mappings, see "Unicode functions (CLI)" on page 5.

Syntax

```
SQLRETURN SQL_API_FN SQLDropDb ( SQLHDBC      hDbc,
                                 SQLCHAR      *szDbName,
                                 SQLINTEGER   cbDbName);
```

Function arguments

Table 44. SQLDropDb function argument

Data type	Argument	Use	Description
SQLHDBC	*hDbc*	input	Connection handle.
SQLCHAR *	*szDbName*	input	Name of the database that is to be dropped.
SQLINTEGER	*cbDbName*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of the function) that is needed to store the *szDbName* argument or to store SQL_NTS if the *szDbName* is null terminated.

Usage

To drop a DB2 database, the CLI application must first attach to the server instance by using the ATTACH keyword. The valid APIs, after connecting to the server instance using ATTACH keyword are SQLCreateDb(), SQLDropDb(), and SQLDisconnect().

Return codes

- SQL_SUCCESS
- SQL_ERROR

Restrictions

- An already connected database cannot be dropped.
- The SQLDropDb() function is not supported for DB2 for IBM i and DB2 for z/OS data servers.

Example

The following example creates and drops DB2 databases on a local server:

```
sqldriverconnect 1 0 "attach=true" -3 50 SQL_DRIVER_NOPROMPT
sqlcreatedb 1 sample1 8 null 0 null 0
sqlcreatedb 1 sample2 8 null 0 null 0
sqldropdb 1 sample1 8
sqldropdb 1 sample2 8
sqldisconnect 1
```

The following example creates and drops DB2 databases on a remote server:

```
sqldriverconnect 1 0  "attach=true;hostname=myhostname;port=9999;
 uid=myuid;pwd=mypwd;protocol=tcpip" -3 50 SQL_DRIVER_NOPROMPT
sqlcreatedb 1 sample1 8 null 0 null 0
sqlcreatedb 1 sample2 8 null 0 null 0
sqldropdb 1 sample1 8
sqldropdb 1 sample2 8
sqldisconnect 1
```

Version information

Last update

This topic was last updated for IBM DB2 Version 9.7, Fix Pack 3.

IBM Data Server Client
Supported in IBM DB2 Database for Linux, UNIX, and Windows

SQLEndTran function (CLI) - End transactions of a connection or an environment

Requests a commit or rollback operation for all active operations on all statements associated with a connection, or for all connections associated with an environment.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

`SQLEndTran()` requests a commit or rollback operation for all active operations on all statements that are associated with a connection, or for all connections that are associated with an environment.

Syntax

```
SQLRETURN   SQLEndTran (
               SQLSMALLINT      HandleType,       /* fHandleType */
               SQLHANDLE        Handle,           /* hHandle */
               SQLSMALLINT      CompletionType);  /* fType */
```

Function arguments

Table 45. SQLEndTran arguments

Data type	Argument	Use	Description
SQLSMALLINT	*HandleType*	Input	*Handle* type identifier. Contains either SQL_HANDLE_ENV if *Handle* is an environment handle, or SQL_HANDLE_DBC if *Handle* is a connection handle.
SQLHANDLE	*Handle*	Input	The handle, of the type that is indicated by *HandleType*, that indicates the scope of the transaction.
SQLSMALLINT	*CompletionType*	Input	One of the following two values: • SQL_COMMIT • SQL_ROLLBACK

Usage

If *HandleType* is SQL_HANDLE_ENV and *Handle* is a valid environment handle,CLI attempts to commit or roll back transactions one at a time, depending on the value of *CompletionType*, on all connections that are in a connected state on that environment. SQL_SUCCESS is returned only if it receives SQL_SUCCESS for each connection. If it receives SQL_ERROR on one or more connections, it returns SQL_ERROR to the application, and the diagnostic information is placed in the diagnostic data structure of the environment. To determine which connections failed during the commit or rollback operation, the application can call `SQLGetDiagRec()` for each connection.

You must not use`SQLEndTran()` when working in a Distributed Unit of Work environment. Use the transaction manager APIs instead.

If *CompletionType* is SQL_COMMIT, SQLEndTran() issues a commit request for all active operations on any statement that is associated with an affected connection. If *CompletionType* is SQL_ROLLBACK, SQLEndTran() issues a rollback request for all active operations on any statement that is associated with an affected connection. If no transactions are active, SQLEndTran() returns SQL_SUCCESS with no effect on any data sources.

To determine how transaction operations affect cursors, an application calls SQLGetInfo() with the SQL_CURSOR_ROLLBACK_BEHAVIOR and SQL_CURSOR_COMMIT_BEHAVIOR options.

If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value equals SQL_CB_DELETE, SQLEndTran() closes and deletes all open cursors on all statements that are associated with the connection, and discards all pending results. SQLEndTran() leaves any statement present in an allocated (unprepared) state; the application can reuse them for subsequent SQL requests or can call SQLFreeStmt() or SQLFreeHandle() with a *HandleType* of SQL_HANDLE_STMT to deallocate them.

If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value equals SQL_CB_CLOSE, SQLEndTran() closes all open cursors on all statements that are associated with the connection. SQLEndTran() leaves any statement present in a prepared state; the application can call SQLExecute() for a statement that is associated with the connection without first calling SQLPrepare().

If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value equals SQL_CB_PRESERVE, SQLEndTran() does not affect open cursors that are associated with the connection. Cursors remain at the row that they pointed to before the call to SQLEndTran().

When autocommit mode is off, calling SQLEndTran() with either SQL_COMMIT or SQL_ROLLBACK when no transaction is active returns SQL_SUCCESS, which indicates that there is no work to be committed or rolled back. Calling SQLEndTran() has no effect on the data source, unless errors that are not related to the transactions occur.

When autocommit mode is on, calling SQLEndTran() with a *CompletionType* of either SQL_COMMIT or SQL_ROLLBACK always returns SQL_SUCCESS, unless errors that are not related to the transactions occur.

When a CLI application is running in autocommit mode, the CLI driver does not pass the statement to the server.

For applications that use the ODBC driver version 3.8 or later, the SQLEnTran function can set the connection to suspended state and returns SQL_ERROR (with SQLSTATE set to HY117). You must set the SQL_ATTR_ODBC_VERSION environment attribute to SQL_OV_ODBC3_80. For more details about necessary conditions to set the connection in a suspended state, see the Microsoft MSDN documentation for the SQLEndTran() at http://msdn.microsoft.com/en-us/library/ms716544(v=vs.85).aspx.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR

- SQL_INVALID_HANDLE

Diagnostics

Table 46. SQLEndTran SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	An informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	The *ConnectionHandle* is not in a connected state.
08007	Connection failure during transaction.	The connection that is associated with the *ConnectionHandle* failed during the execution of the function, and it cannot be determined whether the requested **COMMIT** or **ROLLBACK** occurred before the failure.
40001	Transaction rollback.	The transaction is rolled back due to a resource deadlock with another transaction.
HY000	General error.	An error occurred for which there is no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	An asynchronously executing function was called for a *StatementHandle* that is associated with the *ConnectionHandle* and was still executing when `SQLEndTran()` was called. `SQLExecute()` or `SQLExecDirect()` was called for a *StatementHandle* that is associated with the *ConnectionHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. An exception to this behavior exists for CLI applications that run against a DB2 for z/OS database server. When the connection attribute SQL_ATTR_FORCE_ROLLBACK is turned on, CLI applications can successfully perform `SQLEndTran()` or `SQLTransact()` when *CompletionType* is SQL_ROLLBACK. The StreamPutData configuration keyword must be set to 1 (on).
HY012	Invalid transaction code.	The value that is specified for the argument *CompletionType* is neither SQL_COMMIT nor SQL_ROLLBACK.
HY092	Option type out of range.	The value specified for the argument *HandleType* was neither SQL_HANDLE_ENV nor SQL_HANDLE_DBC.

Restrictions

None.

Example

```
/* commit all active transactions on the connection */
cliRC = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT)

/* ... */

/* rollback all active transactions on the connection */
cliRC = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
```

```
/* ... */

/* rollback all active transactions on all connections
   in this environment */
cliRC = SQLEndTran(SQL_HANDLE_ENV, henv, SQL_ROLLBACK);
```

SQLError function (CLI) - Retrieve error information

In ODBC 3.0, SQLError() has been deprecated and replaced with SQLGetDiagRec() and SQLGetDiagField().

Although this version of CLI continues to support SQLError(), use SQLGetDiagRec() in your CLI programs so that they conform to the latest standards.

Note:

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLErrorW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Migrating to the new function

To read the error diagnostic records for a statement handle, the `SQLError()` function,

```
SQLError(henv, hdbc, hstmt, *szSqlState, *pfNativeError,
         *szErrorMsg, cbErrorMsgMax, *pcbErrorMsg);
```

for example, would be rewritten using the new function as:

```
SQLGetDiagRec(SQL_HANDLE_HSTMT, hstmt, 1, szSqlState, pfNativeError,
              szErrorMsg, cbErrorMsgMax, pcbErrorMsg);
```

SQLExecDirect function (CLI) - Execute a statement directly

Directly executes the specified SQL statement or XQuery expression using the current values of the parameter marker variables if any parameters exist in the statement.

The statement or expression can only be executed once.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

For XQuery expressions, you cannot specify parameter markers in the expression itself. You can, however, use the XMLQUERY function to bind parameter markers to XQuery variables. The values of the bound parameter markers will then be passed to the XQuery expression specified in XMLQUERY for execution.

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLExecDirectW(). Refer to "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

SQLExecDirect function (CLI) - Execute a statement directly

Syntax

```
SQLRETURN   SQLExecDirect (
               SQLHSTMT         StatementHandle,        /* hstmt */
               SQLCHAR          *StatementText,         /* szSqlStr */
               SQLINTEGER       TextLength);            /* cbSqlStr */
```

Function arguments

Table 47. SQLExecDirect arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle. There must not be an open cursor associated with *StatementHandle*.
SQLCHAR *	*StatementText*	input	SQL statement or XQuery expression string.
SQLINTEGER	*TextLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the *StatementText* argument, or SQL_NTS if *StatementText* is null-terminated.

Usage

If the SQL statement text contains vendor escape clause sequences, CLI will first modify the SQL statement text to the appropriate DB2 specific format before submitting it for preparation and execution. If the application does not generate SQL statements that contain vendor escape clause sequences, then it must set the SQL_ATTR_NOSCAN statement attribute to SQL_NOSCAN_ON at the connection level so that CLI does not perform a scan for vendor escape clauses.

The SQL statement can be COMMIT or ROLLBACK if it is called using SQLExecDirect(). Doing so yields the same result as calling SQLEndTran() on the current connection handle.

The SQL statement string can contain parameter markers, however all parameters must be bound before calling SQLExecDirect().

If the SQL statement is a query, or *StatementText* is an XQuery expression, SQLExecDirect() will generate a cursor name, and open the cursor. If the application has used `SQLSetCursorName()` to associate a cursor name with the statement handle, CLI associates the application generated cursor name with the internally generated one.

If a result set is generated, `SQLFetch()` or `SQLFetchScroll()` will retrieve the next row (or rows) of data into bound variables, LOB locators, or LOB file references.

If the SQL statement is a positioned DELETE or a positioned UPDATE, the cursor referenced by the statement must be positioned on a row and must be defined on a separate statement handle under the same connection handle.

There must not already be an open cursor on the statement handle.

If `SQLSetStmtAttr()` has been called with the SQL_ATTR_PARAMSET_SIZE attribute to specify that an array of input parameter values has been bound to each parameter marker, then the application needs to call SQLExecDirect() only once to process the entire array of input parameter values.

If the executed statement returns multiple result sets (one for each set of input parameters), then `SQLMoreResults()` must be used to advance to the next result set when processing on the current result set is completed.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA
- SQL_NO_DATA_FOUND

SQL_NEED_DATA is returned when the application has requested to input data-at-execute parameter values by setting the **StrLen_or_IndPtr* value specified during `SQLBindParameter()` to SQL_DATA_AT_EXEC for one or more parameters.

SQL_NO_DATA_FOUND is returned if the SQL statement is a Searched UPDATE or Searched DELETE and no rows satisfy the search condition.

Diagnostics

Table 48. SQLExecDirect SQLSTATEs

SQLSTATE	Description	Explanation
01504	The UPDATE or DELETE statement does not include a WHERE clause.	*StatementText* contained an UPDATE or DELETE statement which did not contain a WHERE clause. (Function returns SQL_SUCCESS_WITH_INFO or SQL_NO_DATA_FOUND if there were no rows in the table).
01508	Statement disqualified for blocking.	The statement was disqualified for blocking for reasons other than storage.
07001	Wrong number of parameters.	The number of parameters bound to application variables using `SQLBindParameter()` was less than the number of parameter markers in the SQL statement contained in the argument *StatementText*.
07006	Invalid conversion.	Transfer of data between CLI and the application variables would result in an incompatible data conversion.
21S01	Insert value list does not match column list.	*StatementText* contained an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	*StatementText* contained a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
22001	String data right truncation.	A character string assigned to a character type column exceeded the maximum length of the column.
22003	Numeric value out of range.	A numeric value assigned to a numeric type column caused truncation of the whole part of the number, either at the time of assignment or in computing an intermediate result. *StatementText* contained an SQL statement with an arithmetic expression which caused division by zero. **Note:** as a result the cursor state is undefined for DB2 Database for Linux, UNIX, and Windows (the cursor will remain open for other RDBMSs).

Table 48. SQLExecDirect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22005	Error in assignment.	*StatementText* contained an SQL statement with a parameter or literal and the value or LOB locator was incompatible with the data type of the associated table column. The length associated with a parameter value (the contents of the *pcbValue* buffer specified on `SQLBindParameter()`) is not valid. The argument *fSQLType* used in `SQLBindParameter()` or `SQLSetParam()`, denoted an SQL graphic data type, but the deferred length argument (*pcbValue*) contains an odd length value. The length value must be even for graphic data types.
22007	Invalid datetime format.	*StatementText* contained an SQL statement with an invalid datetime format; that is, an invalid string representation or value was specified, or the value was an invalid date, time, or timestamp.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero is invalid.	*StatementText* contained an SQL statement with an arithmetic expression that caused division by zero.
23000	Integrity constraint violation.	The execution of the SQL statement is not permitted because the execution would cause integrity constraint violation in the DBMS.
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.	Results were pending on the *StatementHandle* from a previous query or a cursor associated with the *hstmt* had not been closed.
34000	Invalid cursor name.	*StatementText* contained a Positioned DELETE or a Positioned UPDATE and the cursor referenced by the statement being executed was not open.
37xxx [a]	Invalid SQL syntax.	*StatementText* contained one or more of : • An SQL statement that the connected database server can not prepare • A statement containing a syntax error
40001	Transaction rollback.	The transaction to which this SQL statement belonged was rolled back due to a deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
42xxx	Syntax Error or Access Rule Violation.	**42**5xx indicates the authorization ID does not have permission to execute the SQL statement contained in *StatementText*. Other **42**xxx SQLSTATES indicate a variety of syntax or access problems with the statement.

Table 48. SQLExecDirect SQLSTATEs (continued)

SQLSTATE	Description	Explanation

428A1	Unable to access a file referenced by a host file variable.	This can be raised for any of the listed scenarios. The associated reason code in the text identifies the particular error:
		• 01 - The file name length is invalid, or the file name, the path has an invalid format, or both.
		• 02 - The file option is invalid. It must have one of the listed values:
		`SQL_FILE_READ -read from an existing file`
		`SQL_FILE_CREATE -create a new file for write`
		`SQL_FILE_OVERWRITE -overwrite an existing file. If the file does not exist, create the file.`
		`SQL_FILE_APPEND -append to an existing file. If the file does not exist, create the file.`
		• 03 - The file cannot be found.
		• 04 - The SQL_FILE_CREATE option was specified for a file with the same name as an existing file.
		• 05 - Access to the file was denied. The user does not have permission to open the file.
		• 06 - Access to the file was denied. The file is in use with incompatible modes. Files to be written to are opened in exclusive mode.
		• 07 - Disk full was encountered while writing to the file.
		• 08 - Unexpected end of file encountered while reading from the file.
		• 09 - A media error was encountered while accessing the file.

42895	The value of a host variable in the EXECUTE or OPEN statement cannot be used because of its data type.	The LOB locator type specified on the bind parameter function call does not match the LOB data type of the parameter marker. The argument *fSQLType* used on the bind parameter function specified a LOB locator type but the corresponding parameter marker is not a LOB.
44000	Integrity constraint violation.	*StatementText* contained an SQL statement which contained a parameter or literal. This parameter value was NULL for a column defined as NOT NULL in the associated table column, or a duplicate value was supplied for a column constrained to contain only unique values, or some other integrity constraint was violated.
56084	LOB data is not supported in DRDA®.	LOB columns cannot either be selected or updated when connecting to host or IBM Power Systems™ servers (using DB2 Connect™).
58004	Unexpected system failure.	Unrecoverable system error.
S0001	Database object already exists.	*StatementText* contained a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already existed.
S0002	Database object does not exist.	*StatementText* contained an SQL statement that references a table name or view name which does not exist.
S0011	Index already exists.	*StatementText* contained a CREATE INDEX statement and the specified index name already existed.
S0012	Index not found.	*StatementText* contained a DROP INDEX statement and the specified index name did not exist.

Table 48. SQLExecDirect SQLSTATEs (continued)

SQLSTATE	Description	Explanation
S0021	Column already exists.	*StatementText* contained an ALTER TABLE statement and the column specified in the ADD clause was not unique or identified an existing column in the base table.
S0022	Column not found.	*StatementText* contained an SQL statement that references a column name which does not exist.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY009	Invalid argument value.	*StatementText* was a null pointer.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The argument *TextLength* was less than 1 but not equal to SQL_NTS.
HY092	Option type out of range.	The *FileOptions* argument of a previous `SQLBindFileToParam()` operation was not valid.
HY503	Invalid file name length.	The *fileNameLength* argument value from `SQLBindFileToParam()` was less than 0, but not equal to SQL_NTS.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Note:

a xxx refers to any SQLSTATE with that class code. Example, **37**xxx refers to any SQLSTATE in the **37** class.

Restrictions

None.

Example

```
/* directly execute a statement - end the COMPOUND statement */
cliRC = SQLExecDirect(hstmt, (SQLCHAR *)"SELECT * FROM ORG", SQL_NTS);
```

SQLExecute function (CLI) - Execute a statement

Executes a statement that was successfully prepared using SQLPrepare() on the same statement handle, once or multiple times.

The statement is executed using the current values of any application variables that were bound to parameter markers by SQLBindParameter() or SQLBindFileToParam().

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

Syntax

```
SQLRETURN   SQLExecute (SQLHSTMT       StatementHandle);  /* hstmt */
```

Function arguments

Table 49. SQLExecute arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle. There must not be an open cursor associated with *StatementHandle*.

Usage

The SQL statement string previously prepared on *StatementHandle* using SQLPrepare() may contain parameter markers. All parameters must be bound before calling SQLExecute().

Note: For XQuery expressions, you cannot specify parameter markers in the expression itself. You can, however, use the XMLQUERY function to bind parameter markers to XQuery variables. The values of the bound parameter markers will then be passed to the XQuery expression specified in XMLQUERY for execution.

Once the application has processed the results from the SQLExecute() call, it can execute the statement again with new (or the same) parameter values.

A statement executed by SQLExecDirect() cannot be re-executed by calling SQLExecute(). Only statements prepared with `SQLPrepare()` can be executed and re-executed with `SQLExecute()`.

If the prepared SQL statement is a query or an XQuery expression, SQLExecute() will generate a cursor name, and open the cursor. If the application has used SQLSetCursorName() to associate a cursor name with the statement handle, CLI associates the application generated cursor name with the internally generated one.

To execute a query more than once on a given statement handle, the application must close the cursor by calling SQLCloseCursor() or SQLFreeStmt() with the SQL_CLOSE option. There must not be an open cursor on the statement handle when calling SQLExecute().

If a result set is generated, SQLFetch() or SQLFetchScroll() will retrieve the next row (or rows) of data into bound variables, LOB locators or LOB file references.

If the SQL statement is a positioned DELETE or a positioned UPDATE, the cursor referenced by the statement must be positioned on a row at the time SQLExecute() is called, and must be defined on a separate statement handle under the same connection handle.

If SQLSetStmtAttr() has been called with the SQL_ATTR_PARAMSET_SIZE attribute to specify that an array of input parameter values has been bound to each parameter marker, the application needs to call SQLExecute() only once to process the entire array of input parameter values. If the executed statement returns multiple result sets (one for each set of input parameters), then SQLMoreResults() should be used to advance to the next result set once processing on the current result set is complete.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA
- SQL_NO_DATA_FOUND

SQL_NEED_DATA is returned when the application has requested to input data-at-execute parameter values by setting the **StrLen_or_IndPtr* value specified during `SQLBindParameter()` to SQL_DATA_AT_EXEC for one or more parameters.

SQL_NO_DATA_FOUND is returned if the SQL statement is a searched UPDATE or searched DELETE and no rows satisfy the search condition.

Diagnostics

The SQLSTATEs for SQLExecute() include all those for SQLExecDirect() except for **HY**009, **HY**090 and with the addition of the SQLSTATE in the following table. Any SQLSTATE that `SQLPrepare()` could return can also be returned on a call to `SQLExecute()` as a result of deferred prepare behavior.

Table 50. SQLExecute SQLSTATEs

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The specified *StatementHandle* was not in a prepared state. SQLExecute() was called without first calling SQLPrepare().

Authorization

None.

Example

```
SQLHANDLE hstmt; /* statement handle */
SQLCHAR *stmt = (SQLCHAR *)"DELETE FROM org WHERE deptnumb = ? ";
SQLSMALLINT parameter1 = 0;

/* allocate a statement handle */
cliRC = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

/* ... */

/* prepare the statement */
cliRC = SQLPrepare(hstmt, stmt, SQL_NTS);

/* ... */

/* bind parameter1 to the statement */
cliRC = SQLBindParameter(hstmt,
                         1,
                         SQL_PARAM_INPUT,
                         SQL_C_SHORT,
                         SQL_SMALLINT,
                         0,
                         0,
                         &parameter1,
                         0,
                         NULL);
/* ... */
```

```
parameter1 = 15;

/* execute the statement for parameter1 = 15  */
cliRC = SQLExecute(hstmt);
```

SQLExtendedBind function (CLI) - Bind an array of columns

Binds an array of columns or parameters instead of using repeated calls to SQLBindCol() or SQLBindParameter().

Specification:

- **CLI** 6

Syntax

```
SQLRETURN    SQLExtendedBind (
                              SQLHSTMT          StatementHandle, /* hstmt */
                              SQLSMALLINT       fBindCol,
                              SQLSMALLINT       cRecords,
                              SQLSMALLINT *     pfCType,
                              SQLPOINTER  *     rgbValue,
                              SQLINTEGER  *     cbValueMax,
                              SQLUINTEGER *     puiPrecisionCType,
                              SQLSMALLINT *     psScaleCType,
                              SQLINTEGER  **    pcbValue,
                              SQLINTEGER  **    piIndicator,
                              SQLSMALLINT *     pfParamType,
                              SQLSMALLINT *     pfSQLType,
                              SQLUINTEGER *     pcbColDef,
                              SQLSMALLINT *     pibScale ) ;
```

Function arguments

Table 51. SQLExtendedBind() arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLSMALLINT	*fBindCol*	input	If SQL_TRUE then the result is similar to SQLBindCol(), otherwise, it is similar to SQLBindParameter().
SQLSMALLINT	*cRecords*	input	Number of columns or parameters to bind.
SQLSMALLINT *	*pfCType*	input	Array of values for the application data type.
SQLPOINTER *	*rgbValue*	input	Array of pointers to application data area.
SQLINTEGER *	*cbValueMax*	input	Array of maximum sizes for *rgbValue*.
SQLUINTEGER *	*puiPrecisionCType*	input	Array of decimal precision values. Each value is used only if the application data type of the corresponding record is SQL_C_DECIMAL_IBM.
SQLSMALLINT *	*psScaleCType*	input	Array of decimal scale values. Each value is used only if the application data type of the corresponding record is SQL_C_DECIMAL_IBM.
SQLINTEGER **	*pcbValue*	input	Array of pointers to length values.
SQLINTEGER **	*piIndicator*	input	Array of pointers to indicator values. The *piIndicator* argument allows the constants SQL_UNASSIGNED and SQL_DEFAULT_PARAM to pass through the method, when extended indicator feature is enabled using the SQL_ATTR_EXTENDED_INDICATORS attribute.

Table 51. SQLExtendedBind() arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT *	*pfParamType*	input	Array of parameter types. Only used if *fBindCol* is FALSE. Each row in this array serves the same purpose as the `SQLBindParameter()` argument *InputOutputType*. It can be set to: • SQL_PARAM_INPUT • SQL_PARAM_INPUT_OUTPUT • SQL_PARAM_OUTPUT
SQLSMALLINT *	*pfSQLType*	input	Array of SQL data types. Only used if *fBindCol* is FALSE. Each row in this array serves the same purpose as the `SQLBindParameter()` argument *ParameterType*.
SQLUINTEGER *	*pcbColDef*	input	Array of SQL precision values. Only used if *fBindCol* is FALSE. Each row in this array serves the same purpose as the `SQLBindParameter()` argument *ColumnSize*.
SQLSMALLINT *	*pibScale*	input	Array of SQL scale values. Only used if *fBindCol* is FALSE. Each row in this array serves the same purpose as the `SQLBindParameter()` argument *DecimalDigits*.

Usage

The argument *fBindCol* determines whether this function call is used to associate (bind):

- parameter markers in an SQL statement (as with `SQLBindParameter()`) - *fBindCol* = SQL_FALSE
- columns in a result set (as with `SQLBindCol()`) - *fBindCol* = SQL_TRUE

This function can be used to replace multiple calls to `SQLBindCol()` or `SQLBindParameter()`, however, important differences should be noted. Depending on how the *fBindCol* parameter has been set, the input expected by `SQLExtendedBind()` is similar to either `SQLBindCol()` or `SQLBindParameter()` with the following exceptions:

- When `SQLExtendedBind()` is set to `SQLBindCol()` mode:
  - *targetValuePtr* must be a positive integer that specifies in bytes, the maximum length of the data that will be in the returned column.
- When `SQLExtendedBind()` is set to `SQLBindParameter()` mode:
  - *ColumnSize* must be a positive integer that specifies the maximum length of the target column in bytes, where applicable.
  - *DecimalDigits* must be set to the correct scale for the target column, where applicable.
  - *ValueType* of SQL_C_DEFAULT should not be used.
  - If *ValueType* is a locator type, the corresponding *ParameterType* should be a matching locator type.
  - All *ValueType* to *ParameterType* mappings should be as closely matched as possible to minimize the conversion that CLI must perform.

Each array reference passed to SQLExtendedBind() must contain at least the number of elements indicated by *cRecords*. If the calling application fails to pass in sufficiently large arrays, CLI may attempt to read beyond the end of the arrays resulting in corrupt data or critical application failure.

Each array passed to SQLExtendedBind() is considered to be a deferred argument, which means the values in the array are examined and retrieved at the time of execution. As a result, ensure that each array is in a valid state and contains valid data when CLI executes using the values in the array. Following a successful execution, if a statement needs to be executed again, you do not need to call SQLExtendedBind() a second time if the handles passed to the original call to SQLExtendedBind() still refer to valid arrays.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 52. SQLExtendedBind() SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The conversion from the data value identified by a row in the *pfCType* argument to the data type identified by the *pfParamType* argument is not a meaningful conversion. (For example, conversion from SQL_C_TYPE_DATE to SQL_DOUBLE.)
07009	Invalid descriptor index	The value specified for the argument *cRecords* exceeded the maximum number of columns in the result set.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY003	Program type out of range.	A row in *pfParamType* or *pfSQLType* was not a valid data type or SQL_C_DEFAULT.
HY004	SQL data type out of range.	The value specified for the argument *pfParamType* is not a valid SQL data type.
HY009	Invalid argument value.	The argument *rgbValue* was a null pointer and the argument *cbValueMax* was a null pointer, and *pfParamType* is not SQL_PARAM_OUTPUT.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY021	Inconsistent descriptor information	The descriptor information checked during a consistency check was not consistent.

Table 52. SQLExtendedBind() SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY090	Invalid string or buffer length.	The value specified for the argument *cbValueMax* is less than 1 and the argument the corresponding row in *pfParamType* or *pfSQLType* is either SQL_C_CHAR, SQL_C_BINARY or SQL_C_DEFAULT.
HY093	Invalid parameter number.	The value specified for a row in the argument *pfCType* was less than 1 or greater than the maximum number of parameters supported by the server.
HY094	Invalid scale value.	The value specified for *pfParamType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified for *DecimalDigits* was less than 0 or greater than the value for the argument *pcbColDef* (precision). The value specified for *pfParamType* was SQL_C_TYPE_TIMESTAMP and the value for *pfParamType* was either SQL_CHAR or SQL_VARCHAR and the value for *DecimalDigits* was less than 0 or greater than 9. The value specified for *pfParamType* was SQL_C_TIMESTAMP_EXT and the value for DecimalDigits was less than 0 or greater than 12.
HY104	Invalid precision value.	The value specified for *pfParamType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified by *pcbColDef* was less than 1.
HY105	Invalid parameter type.	*pfParamType* is not one of SQL_PARAM_INPUT, SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT.
HYC00	Driver not capable.	CLI recognizes, but does not support the data type specified in the row in *pfParamType* or *pfSQLType*. A LOB locator C data type was specified, but the connected server does not support LOB data types.

Restrictions

None

SQLExtendedFetch function (CLI) - Extended fetch (fetch array of rows)

In ODBC 3.0, SQLExtendedFetch() has been deprecated and replaced with SQLFetchScroll().

Although this version of CLI continues to support SQLExtendedFetch(), use SQLFetchScroll() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

The statement:

```
SQLExtendedFetch(hstmt, SQL_FETCH_ABSOLUTE, 5, &rowCount, &rowStatus);
```

for example, would be rewritten using the new function as:

```
SQLFetchScroll(hstmt, SQL_FETCH_ABSOLUTE, 5);
```

Note:

The information returned in the *rowCount* and *rowStatus* parameters of `SQLExtendedFetch()` are handled by `SQLFetchScroll()` as follows:

- *rowCount*: `SQLFetchScroll()` returns the number of rows fetched in the buffer pointed to by the SQL_ATTR_ROWS_FETCHED_PTR statement attribute.
- *rowStatus*: `SQLFetchScroll()` returns the array of statuses for each row in the buffer pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute.

SQLExtendedPrepare function (CLI) - Prepare a statement and set statement attributes

Prepares a statement and set a group of statement attributes, all in one call.

Specification:

- **CLI** 6.0

This function can be used in place of a call to `SQLPrepare()` followed by a number of calls to `SQLSetStmtAttr()`.

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLExtendedPrepareW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN SQLExtendedPrepare(
             SQLHSTMT      StatementHandle,  /* hstmt */
             SQLCHAR       *StatementText,   /* pszSqlStmt */
             SQLINTEGER    TextLength,       /* cbSqlStmt */
             SQLINTEGER    cPars,
             SQLSMALLINT   sStmtType,
             SQLINTEGER    cStmtAttrs,
             SQLINTEGER    *piStmtAttr,
             SQLINTEGER    *pvParams );
```

Function arguments

Table 53. SQLExtendedPrepare() arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	Statement handle.
SQLCHAR *	*StatementText*	Input	SQL statement string.
SQLINTEGER	*TextLength*	Input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the *StatementText* argument, or SQL_NTS if *StatementText* is null-terminated.
SQLINTEGER	*cPars*	Input	Number of parameter markers in statement.
SQLSMALLINT	*cStmtType*	Input	Statement type. For possible values see List of cStmtType Values.
SQLINTEGER	*cStmtAttrs*	Input	Number of statement attributes specified on this call.
SQLINTEGER *	*piStmtAttr*	Input	Array of statement attributes to set.
SQLINTEGER *	*pvParams*	Input	Array of corresponding statement attributes values to set.

Usage

The first three arguments of this function are exactly the same as the arguments in `SQLPrepare()`.

There are two requirements when using SQLExtendedPrepare():

1. The SQL statements will not be scanned for ODBC/vendor escape clauses. It behaves as if the SQL_ATTR_NOSCAN statement attribute is set to SQL_NOSCAN. If the SQL statement contains ODBC/vendor escape clauses then SQLExtendedPrepare() cannot be used.
2. You must indicate in advance (through *cPars*) the number of parameter markers that are included in the SQL statement.

The *cPars* argument indicates the number of parameter markers in *StatementText*.

The argument *cStmtType* is used to indicate the type of statement that is being prepared. See List of cStmtType Values for the list of possible values.

The final three arguments are used to indicate a set of statement attributes to use. Set *cStmtAttrs* to the number of statement attributes specified on this call. Create two arrays, one to hold the list of statement attributes, one to hold the value for each. Use these arrays for *piStmtAttr* and *pvParams*.

List of cStmtType Values

The argument *cStmtType* can be set to one of the following values:

- SQL_CLI_STMT_UNDEFINED
- SQL_CLI_STMT_ALTER_TABLE
- SQL_CLI_STMT_CREATE_INDEX
- SQL_CLI_STMT_CREATE_TABLE
- SQL_CLI_STMT_CREATE_VIEW
- SQL_CLI_STMT_DELETE_SEARCHED
- SQL_CLI_STMT_DELETE_POSITIONED
- SQL_CLI_STMT_GRANT
- SQL_CLI_STMT_INSERT
- SQL_CLI_STMT_INSERT_VALUES
- SQL_CLI_STMT_REVOKE
- SQL_CLI_STMT_SELECT
- SQL_CLI_STMT_UPDATE_SEARCHED
- SQL_CLI_STMT_UPDATE_POSITIONED
- SQL_CLI_STMT_CALL
- SQL_CLI_STMT_SELECT_FOR_UPDATE
- SQL_CLI_STMT_WITH
- SQL_CLI_STMT_SELECT_FOR_FETCH
- SQL_CLI_STMT_VALUES
- SQL_CLI_STMT_CREATE_TRIGGER
- SQL_CLI_STMT_SELECT_OPTIMIZE_FOR_NROWS
- SQL_CLI_STMT_SELECT_INTO
- SQL_CLI_STMT_CREATE_PROCEDURE
- SQL_CLI_STMT_CREATE_FUNCTION
- SQL_CLI_STMT_SET_CURRENT_QUERY_OPT

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 54. SQLExtendedPrepare SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01504	The UPDATE or DELETE statement does not include a WHERE clause.	*StatementText* contained an UPDATE or DELETE statement which did not contain a WHERE clause.
01508	Statement disqualified for blocking.	The statement was disqualified for blocking for reasons other than storage.
01S02	Option value changed.	CLI did not support a value specified in **pvParams*, or a value specified in **pvParams* was invalid because of SQL constraints or requirements, so CLI substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication link failure.	The communication link between CLI and the data source to which it was connected failed before the function completed processing.
21S01	Insert value list does not match column list.	*StatementText* contained an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	*StatementText* contained a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
22018	Invalid character value for cast specification.	**StatementText* contained an SQL statement that contained a literal or parameter and the value was incompatible with the data type of the associated table column.
22019	Invalid escape character	The argument *StatementText* contained a LIKE predicate with an ESCAPE in the WHERE clause, and the length of the escape character following ESCAPE was not equal to 1.
22025	Invalid escape sequence	The argument *StatementText* contained "LIKE *pattern value* ESCAPE *escape character*" in the WHERE clause, and the character following the escape character in the pattern value was not one of "%" or "_".
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
34000	Invalid cursor name.	*StatementText* contained a positioned DELETE or a positioned UPDATE and the cursor referenced by the statement being executed was not open.
37xxx [a]	Invalid SQL syntax.	*StatementText* contained one or more of the following complications: • an SQL statement that the connected database server could not prepare • a statement containing a syntax error
40001	Transaction rollback.	The transaction to which this SQL statement belonged was rolled back due to deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.

Table 54. SQLExtendedPrepare SQLSTATEs (continued)

SQLSTATE	Description	Explanation
42xxx [a]	Syntax Error or Access Rule Violation.	**42**5xx indicates the authorization ID does not have permission to execute the SQL statement contained in *StatementText*. Other **42**xxx SQLSTATES indicate a variety of syntax or access problems with the statement.
58004	Unexpected system failure.	Unrecoverable system error.
S0001	Database object already exists.	*StatementText* contained a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already existed.
S0002	Database object does not exist.	*StatementText* contained an SQL statement that references a table name or a view name which did not exist.
S0011	Index already exists.	*StatementText* contained a CREATE INDEX statement and the specified index name already existed.
S0012	Index not found.	*StatementText* contained a DROP INDEX statement and the specified index name did not exist.
S0021	Column already exists.	*StatementText* contained an ALTER TABLE statement and the column specified in the ADD clause was not unique or identified an existing column in the base table.
S0022	Column not found.	*StatementText* contained an SQL statement that references a column name which did not exist.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY009	Invalid argument value.	*StatementText* was a null pointer.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY011	Operation invalid at this time.	The *Attribute* was SQL_ATTR_CONCURRENCY, SQL_ ATTR_CURSOR_TYPE, SQL_ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS and the statement was prepared.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY017	Invalid use of an automatically allocated descriptor handle.	The *Attribute* argument was SQL_ATTR_IMP_ROW_DESC or SQL_ATTR_IMP_PARAM_DESC. The *Attribute* argument was SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and the value in **ValuePtr* was an implicitly allocated descriptor handle.

Table 54. SQLExtendedPrepare SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY024	Invalid attribute value.	Given the specified *Attribute* value, an invalid value was specified in **ValuePtr*. (CLI returns this SQLSTATE only for connection and statement attributes that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE. For all other connection and statement attributes, the driver must verify the value specified in **ValuePtr*.)
HY090	Invalid string or buffer length.	The argument *TextLength* was less than 1, but not equal to SQL_NTS.
HY092	Option type out of range.	The value specified for the argument *Attribute* was not valid for this version of CLI.
HYC00	Driver not capable.	The value specified for the argument *Attribute* was a valid connection or statement attribute for the version of the CLI driver, but was not supported by the data source.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Note:

a xxx refers to any SQLSTATE with that class code. Example, **37**xxx refers to any SQLSTATE in the **37** class.

Note: Not all DBMSs report all of the diagnostic messages at prepare time. If deferred prepare is left on as the default behavior (controlled by the SQL_ATTR_DEFERRED_PREPARE statement attribute), then these errors could occur when the PREPARE is flowed to the server. The application must be able to handle these conditions when calling functions that cause this flow. These functions include `SQLExecute()`, `SQLDescribeParam()`, `SQLNumResultCols()`, `SQLDescribeCol()`, and `SQLColAttribute()`.

Restrictions

When accessing IDS data servers, only IDS data server specific SQLExtendedPrepare() attributes are supported. If any SQLExtendedPrepare() attributes not supported by the IDS data server are used, a "Driver not capable" error is returned.

SQLExtendedProcedures function (CLI) - Get list of procedure names

The `SQLExtendedProcedures()` function returns a list of stored procedure names that are registered at the server, and which match the specified search pattern.

The information is returned in an SQL result set, which you can retrieve by using the same functions that you use to process a result set that is generated by a query.

Specification:

- **CLI** 9.7

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is `SQLExtendedProceduresW()`. For information about ANSI to Unicode function mappings, see “Unicode functions (CLI)” on page 5.

Modules are an extension to the concept of schemas. Applications connecting to DB2 version 9.7 or later data servers can create modules inside their schema and can create procedures inside the modules. The fully qualified name of a procedure in a module would be <SCHEMA NAME>.<MODULE NAME>.<PROCEDURE NAME>. The SQLExtendedProcedures() and SQLExtendedProcedureColumns() functions provide information about modules. These functions are not part of the current ODBC specification. For more information, see "Modules" in *SQL Procedural Languages: Application Enablement and Support*.

Syntax

```
SQLRETURN   SQLExtendedProcedures    (
               SQLHSTMT          StatementHandle,   /* hstmt */
               SQLCHAR           *CatalogName,      /* szProcCatalog */
               SQLSMALLINT       NameLength1,       /* cbProcCatalog */
               SQLCHAR           *SchemaName,       /* szProcSchema */
               SQLSMALLINT       NameLength2,       /* cbProcSchema */
               SQLCHAR           *ProcName,         /* szProcName */
               SQLSMALLINT       NameLength3),      /* cbProcName */
               SQLCHAR           *ProcModule,       /* szProcModule */
               SQLSMALLINT       NameLength4;       /* cbProcModule */
```

Function arguments

Table 55. SQLExtendedProcedures arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	The statement handle.
SQLCHAR *	*CatalogName*	Input	A catalog qualifier of a 3-part table name. If the target DBMS does not support 3-part naming, and *CatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS is returned. Otherwise, this is a valid filter for DBMSs that supports 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.
SQLCHAR *	*SchemaName*	Input	A buffer that can contain a *pattern value* to qualify the result set by schema name. For DB2 for MVS/ESA V 4.1 and later, all the stored procedures are in one schema; the only acceptable value for the *SchemaName* argument is a null pointer. If a value is specified, an empty result set and SQL_SUCCESS are returned. For DB2 Database for Linux, UNIX, and Windows, *SchemaName* can contain a valid pattern value. For more information about valid search patterns, see the catalog functions input arguments.
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*ProcName*	Input	A buffer that can contain a *pattern value* to qualify the result set by table name.

Table 55. SQLExtendedProcedures arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *ProcName*, or SQL_NTS if *ProcName* is null-terminated.
SQLCHAR *	*ProcModule*	Input	A buffer that can contain a *pattern value* to qualify the result set by module name.
SQLSMALLINT	*NameLength4*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *ProcModule*, or SQL_NTS if *ProcModule* is null-terminated.

Usage

The result set that is returned by the `SQLExtendedProcedures()` function contains the columns that are listed in Columns returned by SQLExtendedProcedures in the order given. The rows are ordered by PROCEDURE_CAT, PROCEDURE_SCHEMA, and PROCEDURE_NAME.

In many cases, calls to the `SQLExtendedProcedures()` function map to a complex and thus expensive query against the system catalog, so you should use the calls sparingly, and save the results rather than repeating calls.

Call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns that are supported by the connected DBMS.

If the SQL_ATTR_LONGDATA_COMPAT connection attribute is set, LOB column types are reported as LONG VARCHAR, LONG VARBINARY, or LONG VARGRAPHIC types.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

If the stored procedure is at a DB2 for MVS/ESA V4.1 up to V6 server, the name of the stored procedures must be registered in the server's SYSIBM.SYSPROCEDURES catalog table. For V8 and later servers, the stored procedure must be registered in the server's SYSIBM.SYSROUTINES and SYSIBM.SYSPARAMS catalog tables.

For other versions of DB2 servers that do not provide facilities for a stored procedure catalog, an empty result set is returned.

You can specify *ALL as a value in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. CLI searches on all existing schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL.

Columns returned by SQLExtendedProcedures

Column 1 PROCEDURE_CAT (VARCHAR(128))
The procedure catalog name. The value is NULL if this procedure does not have catalogs.

Column 2 PROCEDURE_SCHEM (VARCHAR(128))
The name of the schema that contains PROCEDURE_NAME.

Column 3 PROCEDURE_NAME (VARCHAR(128) NOT NULL)
The name of the procedure.

Column 4 NUM_INPUT_PARAMS (INTEGER not NULL)
The number of input parameters. INOUT parameters are not counted as part of this number.

To determine information regarding INOUT parameters, examine the COLUMN_TYPE column that is returned by `SQLProcedureColumns()`.

Column 5 NUM_OUTPUT_PARAMS (INTEGER not NULL)
The number of output parameters. INOUT parameters are not counted as part of this number.

To determine information regarding INOUT parameters, examine the COLUMN_TYPE column that is returned by `SQLProcedureColumns()`.

Column 6 NUM_RESULT_SETS (INTEGER not NULL)
The number of result sets that are returned by the procedure.

You should not use this column, it is reserved for future use by ODBC.

Column 7 REMARKS (VARCHAR(254))
Contains the descriptive information about the procedure.

Column 8 PROCEDURE_TYPE (SMALLINT)
Defines the procedure type:
- SQL_PT_UNKNOWN: It cannot be determined whether the procedure returns a value.
- SQL_PT_PROCEDURE: The returned object is a procedure that does not have a return value
- SQL_PT_FUNCTION: The returned object is a function that has a return value.

CLI always returns SQL_PT_PROCEDURE.

Column 9 SPECIFIC_NAME (VARCHAR(128))
The unique specific name of PROCEDURE_NAME.

Column 10 PROCEDURE_MODULE (VARCHAR(128))
The name of the module that contains PROCEDURE_NAME within the schema.

Note:

- The column names that are used by CLI follow the X/Open CLI CAE specification style. The column types, contents, and order are identical to those defined for the `SQLExtendedProcedures()` result set in ODBC.
- If two modules contain procedures that share the same name, the `SQLExtendedProcedures()` function returns details about both procedures.

Return codes

- SQL_ERROR
- SQL_INVALID_HANDLE

- SQL_STILL_EXECUTING
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

Diagnostics

Table 56. SQLExtendedProcedures SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. You can set the timeout period byt using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

If an application is connected to a DB2 server that does not provide support for a stored procedure catalog, or does not provide support for stored procedures, the `SQLExtendedProcedures()` function returns an empty result set.

SQLExtendedProcedureColumns function (CLI) - Get input/output parameter information for a procedure

Returns a list of input and output parameters associated with a stored procedure.

The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Specification:

- **CLI** 9.7

`SQLExtendedProcedureColumns()` returns a list of input and output parameters associated with a stored procedure. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Unicode equivalent: This You can also use this function with the Unicode character set. The corresponding Unicode function is `SQLExtendedProcedureColumnsW()`. For information about ANSI to Unicode function mappings, see "Unicode functions (CLI)" on page 5.

Modules are an extension to the concept of schemas. Applications that connect to DB2 version 9.7 or later data servers can create modules inside their schema, and can create procedures inside the modules. The fully qualified name of a procedure in a module would be <SCHEMA NAME>.<MODULE NAME>.<PROCEDURE NAME>. The `SQLExtendedProcedures()` and `SQLExtendedProcedureColumns()` functions provide information about modules. These functions are not part of the current ODBC specification. See in *SQL Procedural Languages: Application Enablement and Support* for more information.

Syntax

```
SQLRETURN SQLExtendedProcedureColumns(
            SQLHSTMT         StatementHandle,   /* hstmt */
            SQLCHAR          *CatalogName,      /* szProcCatalog */
            SQLSMALLINT      NameLength1,       /* cbProcCatalog */
            SQLCHAR          *SchemaName,       /* szProcSchema */
            SQLSMALLINT      NameLength2,       /* cbProcSchema */
            SQLCHAR          *ProcName,         /* szProcName */
            SQLSMALLINT      NameLength3,       /* cbProcName */
            SQLCHAR          *ColumnName,       /* szColumnName */
            SQLSMALLINT      NameLength4),      /* cbColumnName */
            SQLCHAR          *ProcModule,       /* szProcModule */
            SQLSMALLINT      NameLength5;       /* cbProcModule */
```

Function arguments

Table 57. SQLExtendedProcedureColumns arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	A statement handle.
SQLCHAR *	*CatalogName*	Input	A catalog qualifier of a 3-part table name. If the target DBMS does not support 3-part naming, and *CatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS is returned. Otherwise, this is a valid filter for DBMSs that support 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.

Table 57. SQLExtendedProcedureColumns arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*SchemaName*	Input	A buffer that can contain a *pattern value* to qualify the result set by schema name. For DB2 Database for Linux, UNIX, and Windows, *SchemaName* can contain a valid pattern value. For more information about valid search patterns, see the catalog functions input arguments.
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*ProcName*	Input	A buffer that can contain a *pattern value* to qualify the result set by procedure name.
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *ProcName*, or SQL_NTS if *ProcName* is null-terminated.
SQLCHAR *	*ColumnName*	Input	A buffer that can contain a *pattern value* to qualify the result set by parameter name. Use this argument to further qualify the result set that is already restricted by specifying a non-empty value for ProcName, SchemaName, or both.
SQLSMALLINT	*NameLength4*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *ColumnName*, or SQL_NTS if *ColumnName* is null-terminated.
SQLCHAR *	*ProcModule*	Input	A buffer that can contain a *pattern value* to qualify the result set by parameter name. Use this argument to further qualify the result set that is already restricted by specifying a non-empty value for ProcName, SchemaName, or ColumnName.
SQLSMALLINT	*NameLength5*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *ProcModule*, or SQL_NTS if *ProcModule* is null-terminated.

Usage

The `SQLExtendedProcedureColumns()` function returns the information in a result set, that is ordered by PROCEDURE_CAT, PROCEDURE_SCHEM, PROCEDURE_NAME, COLUMN_TYPE and PROCEDURE_MODULE. Columns returned by SQLExtendedProcedureColumns lists the columns in the result set. Columns that are beyond the last column might be defined in future releases.

In many cases, calls to the `SQLExtendedProcedureColumns()` function map to a complex and thus expensive query against the system catalog, therefore you should use these calls sparingly, and save the results rather than repeating calls.

Call the `SQLGetInfo()` function with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, and COLUMN_NAME columns that are supported by the connected DBMS.

If the SQL_ATTR_LONGDATA_COMPAT connection attribute is set, LOB column types are reported as LONG VARCHAR, LONG VARBINARY or LONG VARGRAPHIC types.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

For versions of other DB2 servers that do not provide facilities for a stored procedure catalog, an empty result set is returned.

CLI returns information about the input, input/output, and output parameters that are associated with the stored procedure, but cannot return descriptor information for any result sets that the stored procedure might return.

You can specify *ALL as a value in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. CLI searches on all existing schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL.

Columns returned by SQLExtendedProcedureColumns

Column 1PROCEDURE_CAT (VARCHAR(128))
The name of the procedure catalog. The value is NULL if this procedure does not have catalogs.

Column 2PROCEDURE_SCHEM (VARCHAR(128))
The name of the schema containing PROCEDURE_NAME.

Column 3PROCEDURE_NAME (VARCHAR(128))
The name of the procedure.

Column 4COLUMN_NAME (VARCHAR(128))
The name of the parameter.

Column 5COLUMN_TYPE (SMALLINT not NULL)
Identifies the type of information that is associated with this row. The values can be:

- SQL_PARAM_TYPE_UNKNOWN : The parameter type is unknown.

  Note: This is not returned.
- SQL_PARAM_INPUT: This parameter is an input parameter.
- SQL_PARAM_INPUT_OUTPUT: This parameter is an input / output parameter.
- SQL_PARAM_OUTPUT: This parameter is an output parameter.
- SQL_RETURN_VALUE: The procedure column is the return value of the procedure.

  Note: This is not returned.
- SQL_RESULT_COL: This parameter is actually a column in the result set.

  Note: This is not returned.

Column 6DATA_TYPE (SMALLINT not NULL)
An SQL data type.

Column 7TYPE_NAME (VARCHAR(128) not NULL)
A character string that represents the name of the data type that corresponds to DATA_TYPE.

Column 8COLUMN_SIZE (INTEGER)
For XML arguments in SQL routines, zero is returned (as XML arguments have no length). For cataloged external routines, however, XML parameters are declared as XML AS CLOB(n), in which case COLUMN_SIZE is the cataloged length, n.

If the DATA_TYPE column value denotes a character or binary string, this column contains the maximum length in SQLCHAR or SQLWCHAR elements. If the DATA_TYPE column value is a graphic (DBCS) string, COLUMN_SIZE is the number of double byte SQLCHAR or SQLWCHAR elements for the parameter.

For date, time, and timestamp data types, COLUMN_SIZE is the total number of SQLCHAR or SQLWCHAR elements that are required to display the value when converted to character data type.

For numeric data types, this is either the total number of digits, or the total number of bits that are allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.

See the table of data type precision.

Column 9BUFFER_LENGTH (INTEGER)
The maximum number of bytes for the associated C buffer to store data from this parameter if SQL_C_DEFAULT is specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` calls. This length excludes any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign.

For XML arguments in SQL routines, zero is returned (as XML arguments have no length). For cataloged external routines, however, XML parameters are declared as XML AS CLOB(n), in which case BUFFER_LENGTH is the cataloged length, n.

See the table of data type length.

Column 10DECIMAL_DIGITS (SMALLINT)
The scale of the parameter. NULL is returned for data types where scale is not applicable.

See the table of data type scale.

Column 11NUM_PREC_RADIX (SMALLINT)
Either 10, 2, or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, and the COLUMN_SIZE column contains the number of bits that are allowed in the parameter.

If DATA_TYPE is an exact numeric data type, this column contains the value 10, and the COLUMN_SIZE and DECIMAL_DIGITS columns contain the number of decimal digits that are allowed for the parameter.

For numeric data types, the DBMS can return a NUM_PREC_RADIX of either 10 or 2.

NULL is returned for data types where radix is not applicable.

Column 12NULLABLE (SMALLINT not NULL)
SQL_NO_NULLS if the parameter does not accept NULL values.

SQL_NULLABLE if the parameter accepts NULL values.

Column 13REMARKS (VARCHAR(254))
Might contain descriptive information about the parameter.

Column 14COLUMN_DEF (VARCHAR)
The default value of the column.

If NULL is specified as the default value, this column is the word NULL, not enclosed in quotation marks. If the default value cannot be represented without truncation, this column contains TRUNCATED, with no enclosing single quotation marks. If no default value is specified, this column is NULL.

You can use the value of COLUMN_DEF to generate a new column definition, except when it contains the value TRUNCATED.

Column 15SQL_DATA_TYPE (SMALLINT not NULL)
The value of the SQL data type as it is displayed in the SQL_DESC_TYPE field of the descriptor. This column is the same as the DATA_TYPE column except for datetime data types (CLI does not support interval data types).

For datetime data types, the SQL_DATA_TYPE field in the result set is SQL_DATETIME, and the SQL_DATETIME_SUB field returns the subcode for the specific datetime data type (SQL_CODE_DATE, SQL_CODE_TIME or SQL_CODE_TIMESTAMP).

Column 16SQL_DATETIME_SUB (SMALLINT)
The subtype code for datetime data types. For all other data types this column returns a NULL (including interval data types, which CLI does not support).

Column 17CHAR_OCTET_LENGTH (INTEGER)
The maximum length in bytes of a character data type column. For all other data types, this column returns a NULL.

Column 18ORDINAL_POSITION (INTEGER NOT NULL)
Contains the ordinal position of the parameter that is given by COLUMN_NAME in this result set. The ORDINAL_POSITION is the ordinal position of the argument to be provided on the CALL statement. The leftmost argument has an ordinal position of 1.

Column 19IS_NULLABLE (Varchar)
- NO if the column does not include NULLs.
- YES if the column can include NULLs.
- Zero-length string if the nullability is unknown.

ISO rules are followed to determine nullability.

An ISO SQL-compliant DBMS cannot return an empty string.

The value that is returned for this column is different than the value that is returned for the NULLABLE column. See the description of the NULLABLE column.

Column 20SPECIFIC_NAME (VARCHAR(128))
The unique specific name of PROCEDURE_NAME.

Column 21PROCEDURE_MODULE (VARCHAR(128))
The name of the module containing PROCEDURE_NAME within the schema.

Note:

- The column names that are used by CLI follow the X/Open CLI CAE specification style. The column types, contents, and order are identical to those defined for the SQLExtendedProcedureColumns() result set in ODBC.
- If two modules contain procedures that share the same name, the SQLExtendedProcedureColumns() function returns details about both procedures.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 58. SQLExtendedProcedureColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
42601	PARMLIST syntax error.	The PARMLIST value that is in the stored procedures catalog table contains a syntax error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called, and before it completed execution, SQLCancel() was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. You can set the timeout period by using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetStmtAttr().

Restrictions

SQLExtendedProcedureColumns() does not return information about the attributes of result sets that might be returned from stored procedures.

If an application is connected to a DB2 server that does not provide support for a stored procedure catalog, or does not provide support for stored procedures, SQLExtendedProcedureColumns() will return an empty result set.

SQLExtendedProcedureColumns() is currently only supported with DB2 Version 9.7 or later.

Example

```
/* get input/output parameter information for a procedure including
extended information */
cliRC = SQLExtendedProcedureColumns(hstmt,
 "CatalogName",
 SQL_NTS,
 "SchemaName",
 SQL_NTS,
 "ProcName",
 SQL_NTS,
 "ColumnName",
 SQL_NTS,
 "ModuleName",
 SQL_NTS );
```

SQLFetch function (CLI) - Fetch next row

Advances the cursor to the next row of the result set, and retrieves any bound columns.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

Columns can be bound to:
- application storage
- LOB locators
- LOB file references

When SQLFetch() is called, the appropriate data transfer is performed, along with any data conversion if conversion was indicated when the column was bound. The columns can also be received individually after the fetch, by calling SQLGetData().

SQLFetch() can only be called after a result set has been generated (using the same statement handle) by either executing a query, calling SQLGetTypeInfo() or calling a catalog function.

Syntax

```
SQLRETURN   SQLFetch (SQLHSTMT      StatementHandle);  /* hstmt */
```

Function arguments

Table 59. SQLFetch arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle

Usage

SQLFetch() can only be called after a result set has been generated on the same statement handle. Before SQLFetch() is called the first time, the cursor is positioned before the start of the result set.

The number of application variables bound with `SQLBindCol()` must not exceed the number of columns in the result set or SQLFetch() will fail.

If `SQLBindCol()` has not been called to bind any columns, then SQLFetch() does not return data to the application, but just advances the cursor. In this case `SQLGetData()` could be called to obtain all of the columns individually. If the cursor is a multirow cursor (that is, the SQL_ATTR_ROW_ARRAY_SIZE is greater than 1), `SQLGetData()` can be called only if SQL_GD_BLOCK is returned when `SQLGetInfo()` is called with an *InfoType* of SQL_GETDATA_EXTENSIONS. (Not all DB2 data sources support SQL_GD_BLOCK.) Data in unbound columns is discarded when SQLFetch() advances the cursor to the next row. For fixed length data types, or small variable length data types, binding columns provides better performance than using `SQLGetData()`.

If LOB values are too large to be retrieved in one fetch, they can be retrieved in pieces by either using `SQLGetData()` (which can be used for any column type), or by binding a LOB locator, and using `SQLGetSubString()`.

If any bound storage buffer is not large enough to hold the data returned by SQLFetch(), the data will be truncated. If character data is truncated, SQL_SUCCESS_WITH_INFO is returned, and an SQLSTATE is generated indicating truncation. The `SQLBindCol()` deferred output argument *pcbValue* will contain the actual length of the column data retrieved from the server. The application should compare the actual output length to the input buffer length (*pcbValue* and *cbValueMax* arguments from `SQLBindCol()`) to determine which character columns have been truncated.

Truncation of numeric data types is reported as a warning if the truncation involves digits to the right of the decimal point. If truncation occurs to the left of the decimal point, an error is returned (refer to the diagnostics section).

Truncation of graphic data types is treated the same as character data types, except that the *rgbValue* buffer is filled to the nearest multiple of two bytes that is still less than or equal to the *cbValueMax* specified in `SQLBindCol()`. Graphic (DBCS) data transferred between CLI and the application is not null-terminated if the C buffer type is SQL_C_CHAR (unless the CLI/ODBC configuration keyword PATCH1 includes the value 64). If the buffer type is SQL_C_DBCHAR, then null-termination of graphic data does occur.

Truncation is also affected by the SQL_ATTR_MAX_LENGTH statement attribute. The application can specify that CLI should not report truncation by calling `SQLSetStmtAttr()` with SQL_ATTR_MAX_LENGTH and a value for the maximum length to return for any one column, and by allocating a *rgbValue* buffer of the

same size (plus the null-terminator). If the column data is larger than the set maximum length, SQL_SUCCESS will be returned and the maximum length, not the actual length will be returned in *pcbValue*.

When all the rows have been retrieved from the result set, or the remaining rows are not needed, `SQLCloseCursor()` or `SQLFreeStmt()` with an option of SQL_CLOSE or SQL_DROP should be called to close the cursor and discard the remaining data and associated resources.

An application cannot mix SQLFetch() with `SQLExtendedFetch()` calls on the same statement handle. It can, however, mix SQLFetch() with `SQLFetchScroll()` calls on the same statement handle. Note that `SQLExtendedFetch()` has been deprecated and replaced with `SQLFetchScroll()`.

Positioning the cursor

When the result set is created, the cursor is positioned before the start of the result set. SQLFetch() fetches the next rowset. It is equivalent to calling `SQLFetchScroll()` with *FetchOrientation* set to SQL_FETCH_NEXT.

The SQL_ATTR_ROW_ARRAY_SIZE statement attribute specifies the number of rows in the rowset. If the rowset being fetched by SQLFetch() overlaps the end of the result set, SQLFetch() returns a partial rowset. That is, if S + R-1 is greater than L, where S is the starting row of the rowset being fetched, R is the rowset size, and L is the last row in the result set, then only the first L-S+1 rows of the rowset are valid. The remaining rows are empty and have a status of SQL_ROW_NOROW.

Refer to the cursor positioning rules of SQL_FETCH_NEXT for `SQLFetchScroll()` for more information.

After SQLFetch() returns, the current row is the first row of the rowset.

Row status array

SQLFetch() sets values in the row status array in the same manner as `SQLFetchScroll()` and `SQLBulkOperations()`. The row status array is used to return the status of each row in the rowset. The address of this array is specified with the SQL_ATTR_ROW_STATUS_PTR statement attribute.

Rows fetched buffer

SQLFetch() returns the number of rows fetched in the rows fetched buffer including those rows for which no data was returned. The address of this buffer is specified with the SQL_ATTR_ROWSFETCHED_PTR statement attribute. The buffer is set by `SQLFetch()` and `SQLFetchScroll()`.

Error handling

Errors and warnings can apply to individual rows or to the entire function. They can be retrieved using the SQLGetDiagField() function.

Errors and Warnings on the Entire Function

If an error applies to the entire function, such as SQLSTATE HYT00 (Timeout expired) or SQLSTATE 24000 (Invalid cursor state), SQLFetch() returns

SQL_ERROR and the applicable SQLSTATE. The contents of the rowset buffers are undefined and the cursor position is unchanged.

If a warning applies to the entire function, SQLFetch() returns SQL_SUCCESS_WITH_INFO and the applicable SQLSTATE. The status records for warnings that apply to the entire function are returned before the status records that apply to individual rows.

Errors and warnings in individual rows

If an error (such as SQLSTATE 22012 (Division by zero)) or a warning (such as SQLSTATE 01004 (Data truncated)) applies to a single row, SQLFetch() returns SQL_SUCCESS_WITH_INFO, unless an error occurs in every row, in which case SQL_ERROR is returned. `SQLFetch()` also:

- Sets the corresponding element of the row status array to SQL_ROW_ERROR for errors or SQL_ROW_SUCCESS_WITH_INFO for warnings.
- Adds zero or more status records containing SQLSTATEs for the error or warning.
- Sets the row and column number fields in the status records. If SQLFetch() cannot determine a row or column number, it sets that number to SQL_ROW_NUMBER_UNKNOWN or SQL_COLUMN_NUMBER_UNKNOWN. If the status record does not apply to a particular column, SQLFetch() sets the column number to SQL_NO_COLUMN_NUMBER.

SQLFetch() returns the status records in row number order. That is, it returns all status records for unknown rows (if any), then all status records for the first row (if any), then all status records for the second row (if any), and so on. The status records for each individual row are ordered according to the normal rules for ordering status records, described in SQLGetDiagField().

Descriptors and SQLFetch

The following sections describe how SQLFetch() interacts with descriptors.

Argument mappings

The driver does not set any descriptor fields based on the arguments of SQLFetch().

Other descriptor fields

The following descriptor fields are used by SQLFetch():

Table 60. Descriptor fields

Descriptor field	Desc.	Location	Set through
SQL_DESC_ARRAY_SIZE	ARD	header	SQL_ATTR_ROW_ARRAY_SIZE statement attribute
SQL_DESC_ARRAY_STATUS_PTR	IRD	header	SQL_ATTR_ROW_STATUS_PTR statement attribute
SQL_DESC_BIND_OFFSET_PTR	ARD	header	SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute
SQL_DESC_BIND_TYPE	ARD	header	SQL_ATTR_ROW_BIND_TYPE statement attribute
SQL_DESC_COUNT	ARD	header	*ColumnNumber* argument of `SQLBindCol()`

Table 60. Descriptor fields (continued)

Descriptor field	Desc.	Location	Set through
SQL_DESC_DATA_PTR	ARD	records	*TargetValuePtr* argument of SQLBindCol()
SQL_DESC_INDICATOR_PTR	ARD	records	*StrLen_or_IndPtr* argument in SQLBindCol()
SQL_DESC_OCTET_LENGTH	ARD	records	*BufferLength* argument in SQLBindCol()
SQL_DESC_OCTET_LENGTH_PTR	ARD	records	*StrLen_or_IndPtr* argument in SQLBindCol()
SQL_DESC_ROWS_PROCESSED_PTR	IRD	header	SQL_ATTR_ROWS_FETCHED_PTR statement attribute
SQL_DESC_TYPE	ARD	records	*TargetType* argument in SQLBindCol()

All descriptor fields can also be set through SQLSetDescField().

Separate length and indicator buffers

Applications can bind a single buffer or two separate buffers to be used to hold length and indicator values. When an application calls SQLBindCol(), SQL_DESC_OCTET_LENGTH_PTR and SQL_DESC_INDICATOR_PTR fields of the ARD are set to the same address, which is passed in the *StrLen_or_IndPtr* argument. When an application calls SQLSetDescField() or SQLSetDescRec(), it can set these two fields to different addresses.

SQLFetch() determines whether the application has specified separate length and indicator buffers. In this case, when the data is not NULL, SQLFetch() sets the indicator buffer to 0 and returns the length in the length buffer. When the data is NULL, SQLFetch() sets the indicator buffer to SQL_NULL_DATA and does not modify the length buffer.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND is returned if there are no rows in the result set, or previous SQLFetch() calls have fetched all the rows from the result set.

If all the rows have been fetched, the cursor is positioned after the end of the result set.

Diagnostics

Table 61. SQLFetch SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data returned for one or more columns was truncated. String values or numeric values are right truncated. (SQL_SUCCESS_WITH_INFO is returned if no error occurred.)
07002	Too many columns.	A column number specified in the binding for one or more columns was greater than the number of columns in the result set.
07006	Invalid conversion.	The data value could not be converted in a meaningful manner to the data type specified by *fCType* in SQLBindCol()

Table 61. SQLFetch SQLSTATEs (continued)

SQLSTATE	Description	Explanation
07009	Invalid descriptor index	Column 0 was bound but bookmarks are not being used (the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF).
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument *pcbValue* in `SQLBindCol()` was a null pointer and the value of the corresponding column is null. There is no means to report SQL_NULL_DATA. The pointer specified for the argument `IndicatorValue` in `SQLBindFileToCol()` was a null pointer and the value of the corresponding LOB column is NULL. There is no means to report SQL_NULL_DATA.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for one or more columns would have caused the whole part of the number to be truncated either at the time of assignment or in computing an intermediate result. A value from an arithmetic expression was returned which resulted in division by zero. **Note:** The associated cursor is undefined if this error is detected by DB2 Database for Linux, UNIX, and Windows. If the error was detected by CLI or by other IBM RDBMSs, the cursor will remain open and continue to advance on subsequent fetch calls.
22005	Error in assignment.	A returned value was incompatible with the data type of binding. A returned LOB locator was incompatible with the data type of the bound column.
22007	Invalid datetime format.	Conversion from character a string to a datetime format was indicated, but an invalid string representation or value was specified, or the value was an invalid date. The value of a date, time, or timestamp does not conform to the syntax for the specified data type.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero is invalid.	A value from an arithmetic expression was returned which resulted in division by zero.
24000	Invalid cursor state.	The previous SQL statement executed on the statement handle was not a query.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.

Table 61. SQLFetch SQLSTATEs (continued)

SQLSTATE	Description	Explanation

428A1	Unable to access a file referenced by a host file variable.	This can be raised for any of the following scenarios. The associated reason code in the text identifies the particular error:
		• 01 - The file name length is invalid, or the file name, the path or both has an invalid format.
		• 02 - The file option is invalid. It must have one of the following values:
		`SQL_FILE_READ -read from an existing file`
		`SQL_FILE_CREATE -create a new file for write`
		`SQL_FILE_OVERWRITE -overwrite an existing file. If the file does not exist, create the file.`
		`SQL_FILE_APPEND -append to an existing file. If the file does not exist, create the file.`
		• 03 - The file cannot be found.
		• 04 - The SQL_FILE_CREATE option was specified for a file with the same name as an existing file.
		• 05 - Access to the file was denied. The user does not have permission to open the file.
		• 06 - Access to the file was denied. The file is in use with incompatible modes. Files to be written to are opened in exclusive mode.
		• 07 - Disk full was encountered while writing to the file.
		• 08 - Unexpected end of file encountered while reading from the file.
		• 09 - A media error was encountered while accessing the file.

54028	The maximum number of concurrent LOB handles has been reached.	Maximum LOB locator assigned. The maximum number of concurrent LOB locators has been reached. A new locator can not be assigned.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	SQLFetch() was called for an StatementHandle after `SQLExtendedFetch()` was called and before `SQLFreeStmt()` had been called with the SQL_CLOSE option. The function was called before calling `SQLPrepare()` or `SQLExecDirect()` for the *StatementHandle*. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Table 61. SQLFetch SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY092	Option type out of range.	The *FileOptions* argument of a previous `SQLBindFileToCol()` operation was not valid.
HYC00	Driver not capable.	CLI or the data source does not support the conversion specified by the combination of the *fCType* in `SQLBindCol()` or `SQLBindFileToCol()` and the SQL data type of the corresponding column. A call to `SQLBindCol()` was made for a column data type which is not supported by CLI.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

Example

```
/* fetch each row and display */
cliRC = SQLFetch(hstmt);
STMT_HANDLE_CHECK(hstmt, hdbc, cliRC);

if (cliRC == SQL_NO_DATA_FOUND)
{
  printf("\n  Data not found.\n");
}
while (cliRC != SQL_NO_DATA_FOUND)
{
  printf("

  /* fetch next row */
  cliRC = SQLFetch(hstmt);
  STMT_HANDLE_CHECK(hstmt, hdbc, cliRC);
}
```

SQLFetchScroll function (CLI) - Fetch rowset and return data for all bound columns

Fetches the specified rowset of data from the result set and returns data for all bound columns.

Rowsets can be specified at an absolute or relative position or by bookmark.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Syntax

```
SQLRETURN   SQLFetchScroll   (SQLHSTMT          StatementHandle,
                              SQLSMALLINT       FetchOrientation,
                              SQLLEN            FetchOffset);
```

Function arguments

Table 62. SQLFetchScroll arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLUSMALLINT	*FetchOrientation*	input	Type of fetch: • SQL_FETCH_NEXT • SQL_FETCH_PRIOR • SQL_FETCH_FIRST • SQL_FETCH_LAST • SQL_FETCH_ABSOLUTE • SQL_FETCH_RELATIVE • SQL_FETCH_BOOKMARK For more information, see Positioning the Cursor.
SQLLEN	*FetchOffset*	input	Number of the row to fetch. The interpretation of this argument depends on the value of the *FetchOrientation* argument. For more information, see Positioning the Cursor.

Usage

Overview

SQLFetchScroll() returns a specified rowset from the result set. Rowsets can be specified by absolute or relative position or by bookmark. SQLFetchScroll() can be called only while a result set exists, that is, after a call that creates a result set and before the cursor over that result set is closed. If any columns are bound, it returns the data in those columns. If the application has specified a pointer to a row status array or a buffer in which to return the number of rows fetched, SQLFetchScroll() returns this information as well. Calls to SQLFetchScroll() can be mixed with calls to SQLFetch() but cannot be mixed with calls to SQLExtendedFetch().

Positioning the cursor

When the result set is created, the cursor is positioned before the start of the result set. SQLFetchScroll() positions the block cursor based on the values of the *FetchOrientation* and *FetchOffset* arguments as shown in the following table. The exact rules for determining the start of the new rowset are shown in the next section.

FetchOrientation
Meaning

SQL_FETCH_NEXT
Return the next rowset. This is equivalent to calling SQLFetch(). SQLFetchScroll() ignores the value of *FetchOffset*.

SQL_FETCH_PRIOR
Return the prior rowset. SQLFetchScroll() ignores the value of *FetchOffset*.

SQL_FETCH_RELATIVE
Return the rowset *FetchOffset* from the start of the current rowset.

SQL_FETCH_ABSOLUTE
Return the rowset starting at row *FetchOffset*.

SQL_FETCH_FIRST
Return the first rowset in the result set. SQLFetchScroll() ignores the value of *FetchOffset*.

SQL_FETCH_LAST
Return the last complete rowset in the result set. SQLFetchScroll() ignores the value of *FetchOffset*.

SQL_FETCH_BOOKMARK
Return the rowset *FetchOffset* rows from the bookmark specified by the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute.

Not all cursors support all of these options. A static forward-only cursor, for example, will only support SQL_FETCH_NEXT. Scrollable cursors, such as keyset cursors, will support all of these options. The SQL_ATTR_ROW_ARRAY_SIZE statement attribute specifies the number of rows in the rowset. If the rowset being fetched by SQLFetchScroll() overlaps the end of the result set, SQLFetchScroll() returns a partial rowset. That is, if S + R-1 is greater than L, where S is the starting row of the rowset being fetched, R is the rowset size, and L is the last row in the result set, then only the first L-S+1 rows of the rowset are valid. The remaining rows are empty and have a status of SQL_ROW_NOROW.

After SQLFetchScroll() returns, the rowset cursor is positioned on the first row of the result set.

Returning data in bound columns

SQLFetchScroll() returns data in bound columns in the same way as SQLFetch().

If no columns are bound, SQLFetchScroll() does not return data but does move the block cursor to the specified position. As with SQLFetch(), you can use SQLGetData() to retrieve the information in this case.

Row status array

The row status array is used to return the status of each row in the rowset. The address of this array is specified with the SQL_ATTR_ROW_STATUS_PTR statement attribute. The array is allocated by the application and must have as many elements as are specified by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute. Its values are set by SQLFetch(), SQLFetchScroll(), or SQLSetPos() (except when they have been called after the cursor has been positioned by SQLExtendedFetch()). If the value of the SQL_ATTR_ROW_STATUS_PTR statement attribute is a null pointer, these functions do not return the row status.

The contents of the row status array buffer are undefined if SQLFetch() or SQLFetchScroll() does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO.

The following values are returned in the row status array.

Row status array value
Description

SQL_ROW_SUCCESS
The row was successfully fetched.

SQL_ROW_SUCCESS_WITH_INFO
The row was successfully fetched. However, a warning was returned about the row.

SQL_ROW_ERROR
An error occurred while fetching the row.

SQL_ROW_ADDED
The row was inserted by SQLBulkOperations(). If the row is fetched again, or is refreshed by SQLSetPos() its status is SQL_ROW_SUCCESS.

This value is not set by SQLFetch() or SQLFetchScroll().

SQL_ROW_UPDATED
The row was successfully fetched and has changed since it was last fetched from this result set. If the row is fetched again from this result set, or is refreshed by SQLSetPos(), the status changes to the row's new status.

SQL_ROW_DELETED
The row has been deleted since it was last fetched from this result set.

SQL_ROW_NOROW
The rowset overlapped the end of the result set and no row was returned that corresponded to this element of the row status array.

Rows fetched buffer

The rows fetched buffer is used to return the number of rows fetched, including those rows for which no data was returned because an error occurred while they were being fetched. In other words, it is the number of rows for which the value in the row status array is not SQL_ROW_NOROW. The address of this buffer is specified with the SQL_ATTR_ROWS_FETCHED_PTR statement attribute. The buffer is allocated by the application. It is set by SQLFetch() and SQLFetchScroll(). If the value of the SQL_ATTR_ROWS_FETCHED_PTR statement attribute is a null pointer, these functions do not return the number of rows fetched. To determine the number of the current row in the result set, an application can call SQLGetStmtAttr() with the SQL_ATTR_ROW_NUMBER attribute.

The contents of the rows fetched buffer are undefined if SQLFetch() or SQLFetchScroll() does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, except when SQL_NO_DATA is returned, in which case the value in the rows fetched buffer is set to 0.

Error handling

SQLFetchScroll() returns errors and warnings in the same manner as SQLFetch().

Descriptors and SQLFetchScroll()

SQLFetchScroll() interacts with descriptors in the same manner as SQLFetch().

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise. If an error occurs on a single column, SQLGetDiagField() can be called with a *DiagIdentifier* of SQL_DIAG_COLUMN_NUMBER to determine the column the error occurred on; and SQLGetDiagField() can be called with a *DiagIdentifier* of SQL_DIAG_ROW_NUMBER to determine the row containing that column.

Table 63. SQLFetchScroll SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	String or binary data returned for a column resulted in the truncation of non-blank character or non-NULL binary data. String values are right truncated. (Function returns SQL_SUCCESS_WITH_INFO.)
01S01	Error in row.	An error occurred while fetching one or more rows. (Function returns SQL_SUCCESS_WITH_INFO.) (This SQLSTATE is only returned when connected to CLI v2.)
01S06	Attempt to fetch before the result set returned the first rowset.	The requested rowset overlapped the start of the result set when the current position was beyond the first row, and either *FetchOrientation* was SQL_PRIOR, or *FetchOrientation* was SQL_RELATIVE with a negative *FetchOffset* whose absolute value was less than or equal to the current SQL_ATTR_ROW_ARRAY_SIZE. (Function returns SQL_SUCCESS_WITH_INFO.)
01S07	Fractional truncation.	The data returned for a column was truncated. For numeric data types, the fractional part of the number was truncated. For time or timestamp data types, the fractional portion of the time was truncated.
07002	Too many columns.	A column number specified in the binding for one or more columns was greater than the number of columns in the result set.
07006	Invalid conversion.	A data value of a column in the result set could not be converted to the C data type specified by *TargetType* in SQLBindCol().
07009	Invalid descriptor index.	Column 0 was bound and the SQL_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.
08S01	Communication link failure.	The communication link between CLI and the data source to which it was connected failed before the function completed processing.
22001	String data right truncation.	A variable-length bookmark returned for a row was truncated.
22002	Invalid output or indicator buffer specified.	NULL data was fetched into a column whose *StrLen_or_IndPtr* set by SQLBindCol() (or SQL_DESC_INDICATOR_PTR set by SQLSetDescField() or SQLSetDescRec()) was a null pointer.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for one or more bound columns would have caused the whole (as opposed to fractional) part of the number to be truncated.
22007	Invalid datetime format.	A character column in the result set was bound to a date, time, or timestamp C structure, and a value in the column was an invalid date, time, or timestamp.
22012	Division by zero is invalid.	A value from an arithmetic expression was returned which resulted in division by zero.

Table 63. SQLFetchScroll SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22018	Invalid character value for cast specification.	A character column in the result set was bound to a character C buffer and the column contained a character for which there was no representation in the character set of the buffer. A character column in the result set was bound to an approximate numeric C buffer and a value in the column could not be cast to a valid approximate numeric value. A character column in the result set was bound to an exact numeric C buffer and a value in the column could not be cast to a valid exact numeric value. A character column in the result set was bound to a datetime C buffer and a value in the column could not be cast to a valid datetime value.
24000	Invalid cursor state.	The *StatementHandle* was in an executed state but no result set was associated with the *StatementHandle*.
40001	Transaction rollback.	The transaction in which the fetch was executed was terminated to prevent deadlock.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The specified *StatementHandle* was not in an executed state. The function was called without first calling SQLExecDirect(), SQLExecute(), or a catalog function. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. SQLExecute() or SQLExecDirect() was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. SQLFetchScroll() was called for a *StatementHandle* after SQLExtendedFetch() was called and before SQLFreeStmt() with SQL_CLOSE was called.
HY106	Fetch type out of range.	The value specified for the argument *FetchOrientation* was invalid. The argument *FetchOrientation* was SQL_FETCH_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF. The value of the SQL_CURSOR_TYPE statement attribute was SQL_CURSOR_FORWARD_ONLY and the value of argument *FetchOrientation* was not SQL_FETCH_NEXT.

Table 63. SQLFetchScroll SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY107	Row value out of range.	The value specified with the SQL_ATTR_CURSOR_TYPE statement attribute was SQL_CURSOR_KEYSET_DRIVEN, but the value specified with the SQL_ATTR_KEYSET_SIZE statement attribute was greater than 0 and less than the value specified with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.
HY111	Invalid bookmark value.	The argument *FetchOrientation* was SQL_FETCH_BOOKMARK and the bookmark pointed to by the value in the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute was not valid or was a null pointer.
HYC00	Driver not capable.	The specified fetch type is not supported. The conversion specified by the combination of the *TargetType* in SQLBindCol() and the SQL data type of the corresponding column is not supported.

Restrictions

None.

Example

```
/* fetch the rowset: row15, row16, row17, row18, row19 */
printf("\n  Fetch the rowset: row15, row16, row17, row18, row19.\n");

/* fetch the rowset and return data for all bound columns */
cliRC = SQLFetchScroll(hstmt, SQL_FETCH_ABSOLUTE, 15);
STMT_HANDLE_CHECK(hstmt, hdbc, cliRC);

/* call SQLFetchScroll with SQL_FETCH_RELATIVE offset 3 */
printf("  SQLFetchScroll with SQL_FETCH_RELATIVE offset 3.\n");
printf("    COL1          COL2         \n");
printf("    ------------  -------------\n");

/* fetch the rowset and return data for all bound columns */
cliRC = SQLFetchScroll(hstmt, SQL_FETCH_RELATIVE, 3);
```

Cursor positioning rules for SQLFetchScroll() (CLI)

The following sections describe the exact rules for each value of *FetchOrientation*. These rules use the following notation:

FetchOrientation
Meaning

Before start
The block cursor is positioned before the start of the result set. If the first row of the new rowset is before the start of the result set, `SQLFetchScroll()` returns SQL_NO_DATA.

After end
The block cursor is positioned after the end of the result set. If the first row of the new rowset is after the end of the result set, `SQLFetchScroll()` returns SQL_NO_DATA.

CurrRowsetStart
The number of the first row in the current rowset.

LastResultRow
The number of the last row in the result set.

RowsetSize
The rowset size.

FetchOffset
The value of the *FetchOffset* argument.

BookmarkRow
The row corresponding to the bookmark specified by the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute.

SQL_FETCH_NEXT rules:

Table 64. SQL_FETCH_NEXT rules:

Condition	First row of new rowset
Before start	1
CurrRowsetStart + RowsetSize <= LastResultRow	CurrRowsetStart + RowsetSize
CurrRowsetStart + RowsetSize > LastResultRow	After end
After end	After end

SQL_FETCH_PRIOR rules:

Table 65. SQL_FETCH_PRIOR rules:

Condition	First row of new rowset
Before start	Before start
CurrRowsetStart = 1	Before start
1 < CurrRowsetStart <= RowsetSize	1 [a]
CurrRowsetStart > RowsetSize	CurrRowsetStart - RowsetSize
After end AND LastResultRow < RowsetSize	1 [a]
After end AND LastResultRow >= RowsetSize	LastResultRow - RowsetSize + 1

- **a** `SQLFetchScroll()` returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first rowset.) and SQL_SUCCESS_WITH_INFO.

SQL_FETCH_RELATIVE rules:

Table 66. SQL_FETCH_RELATIVE rules:

Condition	First row of new rowset
(Before start AND *FetchOffset* > 0) OR (After end AND *FetchOffset* 0)	-- [a]
Before start AND *FetchOffset* <= 0	Before start
CurrRowsetStart = 1 AND FetchOffset < 0	Before start
CurrRowsetStart > 1 AND CurrRowsetStart + *FetchOffset* <1 AND |*FetchOffset*| > RowsetSize	Before start
CurrRowsetStart > 1 AND CurrRowsetStart + *FetchOffset* <1 AND |*FetchOffset*| <= RowsetSize	1 [b]
1 <= CurrRowsetStart + *FetchOffset* <= LastResultRow	CurrRowsetStart + *FetchOffset*
CurrRowsetStart + *FetchOffset* > LastResultRow	After end

Table 66. SQL_FETCH_RELATIVE rules: (continued)

Condition	First row of new rowset
After end AND *FetchOffset* >= 0	After end

- **a** `SQLFetchScroll()` returns the same rowset as if it was called with *FetchOrientation* set to SQL_FETCH_ABSOLUTE. For more information, see the SQL_FETCH_ABSOLUTE section.
- **b** `SQLFetchScroll()` returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first rowset.) and SQL_SUCCESS_WITH_INFO.

SQL_FETCH_ABSOLUTE rules:

Table 67. SQL_FETCH_ABSOLUTE rules:

Condition	First row of new rowset
FetchOffset <0 AND |*FetchOffset*| <= LastResultRow	LastResultRow + *FetchOffset* + 1
FetchOffset <0 AND |*FetchOffset*| > LastResultRow AND |FetchOffset| > RowsetSize	Before start
FetchOffset <0 AND |*FetchOffset*| > LastResultRow AND |FetchOffset| <= RowsetSize	1 [a]
FetchOffset = 0	Before start
1 <= *FetchOffset* <= LastResultRow	*FetchOffset*
FetchOffset > LastResultRow	After end

- **a** `SQLFetchScroll()` returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first rowset.) and SQL_SUCCESS_WITH_INFO.

SQL_FETCH_FIRST rules:

Table 68. SQL_FETCH_FIRST rules:

Condition	First row of new rowset
Any	1

SQL_FETCH_LAST rules:

Table 69. SQL_FETCH_LAST rules:

Condition	First row of new rowset
RowsetSize = LastResultRow	LastResultRow - RowsetSize + 1
RowsetSize > LastResultRow	1

SQL_FETCH_BOOKMARK rules:

Table 70. SQL_FETCH_BOOKMARK rules:

Condition	First row of new rowset
BookmarkRow + *FetchOffset* <1	Before start
1 <= BookmarkRow + *FetchOffset* <= LastResultRow	BookmarkRow +*FetchOffset*
BookmarkRow + *FetchOffset* > LastResultRow	After end

SQLForeignKeys function (CLI) - Get the list of foreign key columns

Returns information about foreign keys for the specified table.

The information is returned in an SQL result set which can be processed using the same functions that are used to retrieve a result generated by a query.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

The `SQLForeignKeys()` function returns information about foreign keys for the specified table. The information is returned in an SQL result set which you can process by using the same functions that you use to retrieve a result that is generated by a query.

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is `SQLForeignKeysW()`. See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLForeignKeys   (
                SQLHSTMT          StatementHandle,  /* hstmt */
                SQLCHAR           *PKCatalogName,   /* szPkCatalogName */
                SQLSMALLINT       NameLength1,      /* cbPkCatalogName */
                SQLCHAR           *PKSchemaName,    /* szPkSchemaName */
                SQLSMALLINT       NameLength2,      /* cbPkSchemaName */
                SQLCHAR           *PKTableName,     /* szPkTableName */
                SQLSMALLINT       NameLength3,      /* cbPkTableName */
                SQLCHAR           *FKCatalogName,   /* szFkCatalogName */
                SQLSMALLINT       NameLength4,      /* cbFkCatalogName */
                SQLCHAR           *FKSchemaName,    /* szFkSchemaName */
                SQLSMALLINT       NameLength5,      /* cbFkSchemaName */
                SQLCHAR           *FKTableName,     /* szFkTableName */
                SQLSMALLINT       NameLength6);     /* cbFkTableName */
```

Function arguments

Table 71. SQLForeignKeys arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	The statement handle.
SQLCHAR *	*PKCatalogName*	Input	The catalog qualifier of the 3-part primary key table name. If the target DBMS does not support 3-part naming, and *PKCatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS is returned. Otherwise, this is a valid filter for DBMSs that support 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *PKCatalogName*, or SQL_NTS if *PKCatalogName* is null-terminated.
SQLCHAR *	*PKSchemaName*	Input	The schema qualifier of the primary key table.

Table 71. SQLForeignKeys arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *PKSchemaName*, or SQL_NTS if *PKSchemaName* is null-terminated.
SQLCHAR *	*PKTableName*	Input	The name of the table name that contains the primary key.
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *PKTableName*, or SQL_NTS if *PKTableName* is null-terminated.
SQLCHAR *	*FKCatalogName*	Input	The catalog qualifier of the 3-part foreign key table name. If the target DBMS does not support 3-part naming, and *FKCatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS is returned. Otherwise, this is a valid filter for DBMSs that support 3-part naming.
SQLSMALLINT	*NameLength4*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *FKCatalogName*, or SQL_NTS if *FKCatalogName* is null-terminated.
SQLCHAR *	*FKSchemaName*	Input	The schema qualifier of the table that contains the foreign key.
SQLSMALLINT	*NameLength5*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *FKSchemaName*, or SQL_NTS if *FKSchemaName* is null-terminated.
SQLCHAR *	*FKTableName*	Input	The name of the table that contains the foreign key.
SQLSMALLINT	*NameLength6*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *FKTableName*, or SQL_NTS if *FKTableName* is null-terminated.

Usage

If *PKTableName* contains a table name, and *FKTableName* is an empty string, the `SQLForeignKeys()` function returns a result set that contains the primary key of the specified table and all of the foreign keys (in other tables) that refer to it.

If *FKTableName* contains a table name, and *PKTableName* is an empty string, the `SQLForeignKeys()` function returns a result set that contains all of the foreign keys in the specified table and the primary keys (in other tables) to which they refer.

If both *PKTableName* and *FKTableName* contain table names, the `SQLForeignKeys()` function returns the foreign keys in the table that are specified in *FKTableName*, which refer to the primary key of the table that is specified in *PKTableName*. There should be one key at the most.

If the schema qualifier argument that is associated with a table name is not specified, the schema name defaults to the table name that is currently in effect for the current connection.

Columns Returned by SQLForeignKeys lists the columns of the result set that is generated by the `SQLForeignKeys()` call. If the foreign keys that are associated with a primary key are requested, the result set is ordered by FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and ORDINAL_POSITION. If the primary keys that are associated with a foreign key are requested, the result set is ordered by PKTABLE_CAT, PKTABLE_SCHEM, PKTABLE_NAME, and ORDINAL_POSITION.

Call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine the actual lengths of the associated TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns that are supported by the connected DBMS.

You can specify *ALL as a value in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. CLI searches on all existing schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns that are returned by SQLForeignKeys

Column 1 PKTABLE_CAT (VARCHAR(128))
Name of the catalog for PKTABLE_NAME. The value is NULL if this table does not have catalogs.

Column 2 PKTABLE_SCHEM (VARCHAR(128))
Name of the schema containing PKTABLE_NAME.

Column 3 PKTABLE_NAME (VARCHAR(128) not NULL)
Name of the table containing the primary key.

Column 4 PKCOLUMN_NAME (VARCHAR(128) not NULL)
Primary key column name.

Column 5 FKTABLE_CAT (VARCHAR(128))
Name of the catalog for FKTABLE_NAME. The value is NULL if this table does not have catalogs.

Column 6 FKTABLE_SCHEM (VARCHAR(128))
Name of the schema containing FKTABLE_NAME.

Column 7 FKTABLE_NAME (VARCHAR(128) not NULL)
Name of the table containing the foreign key.

Column 8 FKCOLUMN_NAME (VARCHAR(128) not NULL)
Foreign key column name.

Column 9 KEY_SEQ (SMALLINT not NULL)
Ordinal position of the column in the key, starting at 1.

Column 10 UPDATE_RULE (SMALLINT)
Action to be applied to the foreign key when the SQL operation is UPDATE:
- SQL_RESTRICT
- SQL_NO_ACTION

The update rule for IBM DB2 DBMSs is always either RESTRICT or SQL_NO_ACTION. However, ODBC applications might encounter the listed UPDATE_RULE values when connected to RDBMSs that are not provided by IBM:
- SQL_CASCADE
- SQL_SET_NULL

Column 11 DELETE_RULE (SMALLINT)
Action to be applied to the foreign key when the SQL operation is DELETE:
- SQL_CASCADE
- SQL_NO_ACTION
- SQL_RESTRICT
- SQL_SET_DEFAULT
- SQL_SET_NULL

Column 12 FK_NAME (VARCHAR(128))
Foreign key identifier. NULL if not applicable to the data source.

Column 13 PK_NAME (VARCHAR(128))
Primary key identifier. NULL if not applicable to the data source.

Column 14 DEFERRABILITY (SMALLINT)
One of:
- SQL_INITIALLY_DEFERRED
- SQL_INITIALLY_IMMEDIATE
- SQL_NOT_DEFERRABLE

Note: The column names that are used by CLI follow the X/Open CLI CAE specification style. The column types, contents, and order are identical to those defined for the `SQLForeignKeys()` result set in ODBC.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 72. SQLForeignKeys SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY009	Invalid argument value.	The arguments *PKTableName* and *FKTableName* were both NULL pointers.

Table 72. SQLForeignKeys SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while in a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS. The length of the table or owner name is greater than the maximum length that is supported by the server.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. You can set the timeout period by using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

Example

```
/* get the list of foreign key columns */
cliRC = SQLForeignKeys(hstmt,
                       NULL,
                       0,
                       tbSchema,
                       SQL_NTS,
                       tbName,
                       SQL_NTS,
                       NULL,
                       0,
                       NULL,
                       SQL_NTS,
                       NULL,
                       SQL_NTS);
```

SQLFreeConnect function (CLI) - Free connection handle

In ODBC 3.0, SQLFreeConnect() has been deprecated and replaced with SQLFreeHandle().

Although this version of CLI continues to support SQLFreeConnect(), use SQLFreeHandle() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

The statement:

```
SQLFreeConnect(hdbc);
```

for example, would be rewritten using the new function as:

```
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
```

SQLFreeEnv function (CLI) - Free environment handle

In ODBC 3.0, SQLFreeEnv() has been deprecated and replaced with SQLFreeHandle().

Although this version of CLI continues to support SQLFreeEnv(), use SQLFreeHandle() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

The statement:

```
SQLFreeEnv(henv);
```

for example, would be rewritten using the new function as:

```
SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

SQLFreeHandle function (CLI) - Free handle resources

Frees resources associated with a specific environment, connection, statement, or descriptor handle.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Note: This function is a generic function for freeing resources. It replaces the ODBC 2.0 functions SQLFreeConnect() (for freeing a connection handle), and SQLFreeEnv() (for freeing an environment handle). SQLFreeHandle() also replaces the ODBC 2.0 function SQLFreeStmt() (with the SQL_DROP Option) for freeing a statement handle.

Syntax

```
SQLRETURN   SQLFreeHandle (
               SQLSMALLINT       HandleType,  /* fHandleType */
               SQLHANDLE         Handle);     /* hHandle */
```

Function arguments

Table 73. SQLFreeHandle arguments

Data type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	The type of handle to be freed by SQLFreeHandle(). Must be one of the following values: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC If *HandleType* is not one of SQL_HANDLE_ENV, SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC value, SQLFreeHandle() returns SQL_INVALID_HANDLE.
SQLHANDLE	*Handle*	input	The handle to be freed.

Usage

SQLFreeHandle() is used to free handles for environments, connections, statements, and descriptors.

An application should not use a handle after it has been freed; CLI does not check the validity of a handle in a function call.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

If SQLFreeHandle() returns SQL_ERROR, the handle is still valid.

Diagnostics

Table 74. SQLFreeHandle SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication link failure.	The *HandleType* argument was SQL_HANDLE_DBC, and the communication link between CLI and the data source to which it was trying to connect failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.

Table 74. SQLFreeHandle SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The *HandleType* argument was SQL_HANDLE_ENV, and at least one connection was in an allocated or connected state. SQLDisconnect() and SQLFreeHandle() with a *HandleType* of SQL_HANDLE_DBC must be called for each connection before calling SQLFreeHandle() with a *HandleType* of SQL_HANDLE_ENV. The *HandleType* argument was SQL_HANDLE_DBC, and the function was called before calling SQLDisconnect() for the connection. The *HandleType* argument was SQL_HANDLE_STMT; an asynchronously executing function was called on the statement handle; and the function was still executing when this function was called. The *HandleType* argument was SQL_HANDLE_STMT; SQLExecute() or SQLExecDirect() was called with the statement handle, and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) All subsidiary handles and other resources were not released before SQLFreeHandle() was called.
HY013	Unexpected memory handling error.	The *HandleType* argument was SQL_HANDLE_STMT or SQL_HANDLE_DESC, and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY017	Invalid use of an automatically allocated descriptor handle.	The *Handle* argument was set to the handle for an automatically allocated descriptor or an implementation descriptor.

Restrictions

None.

Example

```
/* free the statement handle */
cliRC = SQLFreeHandle(SQL_HANDLE_STMT, hstmt2);
SRV_HANDLE_CHECK_SETTING_SQLRC_AND_MSG(SQL_HANDLE_STMT,
                                       hstmt2,
                                       cliRC,
                                       henv,
                                       hdbc,
                                       pOutSqlrc,
                                       outMsg,
                                       "SQLFreeHandle");
/* ... */
/* free the database handle */
cliRC = SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
SRV_HANDLE_CHECK_SETTING_SQLRC_AND_MSG(SQL_HANDLE_DBC,
                                       hdbc,
                                       cliRC,
                                       henv,
                                       hdbc,
                                       pOutSqlrc,
                                       outMsg,
                                       "SQLFreeHandle");

/* free the environment handle */
cliRC = SQLFreeHandle(SQL_HANDLE_ENV, henv);
SRV_HANDLE_CHECK_SETTING_SQLRC_AND_MSG(SQL_HANDLE_ENV,
                                       henv,
```

```
                                        cliRC,
                                        henv,
                                        hdbc,
                                        pOutSqlrc,
                                        outMsg,
                                        "SQLFreeHandle");
```

SQLFreeStmt function (CLI) - Free (or reset) a statement handle

Ends processing on the statement referenced by the statement handle.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

Use this function to:

- Close a cursor and discard all pending results
- Disassociate (reset) parameters from application variables and LOB file references
- Unbind columns from application variables and LOB file references
- Drop the statement handle and free the CLI resources associated with the statement handle.

SQLFreeStmt() is called after executing an SQL statement and processing the results.

Syntax

```
SQLRETURN   SQLFreeStmt (SQLHSTMT          StatementHandle,   /* hstmt */
                         SQLUSMALLINT      Option);           /* fOption */
```

Function arguments

Table 75. SQLFreeStmt arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLUSMALLINT	*Option*	input	Option which specifies the manner of freeing the statement handle. The option must have one of the following values: • SQL_CLOSE • SQL_DROP • SQL_UNBIND • SQL_RESET_PARAMS

Usage

SQLFreeStmt() can be called with the following options:

SQL_CLOSE

The cursor (if any) associated with the statement handle (*StatementHandle*) is closed and all pending results are discarded. The application can reopen the cursor by calling SQLExecute() with the same or different values in the application variables (if any) that are bound to *StatementHandle*. The cursor name is retained until the statement handle is dropped or a subsequent call

to SQLGetCursorName() is successful. If no cursor has been associated with the statement handle, this option has no effect (no warning or error is generated).

SQLCloseCursor() can also be used to close a cursor.

SQL_DROP
CLI resources associated with the input statement handle are freed, and the handle is invalidated. The open cursor, if any, is closed and all pending results are discarded.

This option has been replaced with a call to SQLFreeHandle() with the *HandleType* set to SQL_HANDLE_STMT. Although this version of CLI continues to support this option, begin using SQLFreeHandle() in your CLI programs so that they conform to the latest standards.

SQL_UNBIND
Sets the SQL_DESC_COUNT field of the ARD (Application Row Descriptor) to 0, releasing all column buffers bound by SQLBindCol() or SQLBindFileToCol() for the given *StatementHandle*. This does not unbind the bookmark column; to do that, the SQL_DESC_DATA_PTR field of the ARD for the bookmark column is set to NULL. Note that if this operation is performed on an explicitly allocated descriptor that is shared by more than one statement, the operation will affect the bindings of all statements that share the descriptor.

SQL_RESET_PARAMS
Sets the SQL_DESC_COUNT field of the APD (Application Parameter Descriptor) to 0, releasing all parameter buffers set by SQLBindParameter() or SQLBindFileToParam() for the given *StatementHandle*. Note that if this operation is performed on an explicitly allocated descriptor that is shared by more than one statement, this operation will affect the bindings of all the statements that share the descriptor.

SQLFreeStmt() has no effect on LOB locators, call SQLExecDirect() with the FREE LOCATOR statement to free a locator.

It is possible to reuse a statement handle to execute a different statement:

- If the handle was associated with a query, catalog function or SQLGetTypeInfo(), you must close the cursor.
- If the handle was bound with a different number or type of parameters, the parameters must be reset.
- If the handle was bound with a different number or type of column bindings, the columns must be unbound.

Alternatively you may drop the statement handle and allocate a new one.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

SQL_SUCCESS_WITH_INFO is not returned if *Option* is set to SQL_DROP, as there would be no statement handle to use when SQLGetDiagRec() or SQLGetDiagField() is called.

Diagnostics

Table 76. SQLFreeStmt SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
HY092	Option type out of range.	The value specified for the argument *Option* was not SQL_CLOSE, SQL_DROP, SQL_UNBIND, or SQL_RESET_PARAMS.
HY506	Error closing a file.	Error encountered while trying to close a temporary file.

Authorization

None.

Example

```
/* free the statement handle */
cliRC = SQLFreeStmt(hstmt, SQL_UNBIND);
rc = HandleInfoPrint(SQL_HANDLE_STMT, hstmt, cliRC, __LINE__, __FILE__);
if (rc != 0)
{
  return 1;
}

/* free the statement handle */
cliRC = SQLFreeStmt(hstmt, SQL_RESET_PARAMS);
rc = HandleInfoPrint(SQL_HANDLE_STMT, hstmt, cliRC, __LINE__, __FILE__);
if (rc != 0)
{
  return 1;
}

/* free the statement handle */
cliRC = SQLFreeStmt(hstmt, SQL_CLOSE);
rc = HandleInfoPrint(SQL_HANDLE_STMT, hstmt, cliRC, __LINE__, __FILE__);
if (rc != 0)
{
  return 1;
}
```

SQLGetConnectAttr function (CLI) - Get current attribute setting

Returns the current setting of a connection attribute.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLGetConnectAttrW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLGetConnectAttr(SQLHDBC          ConnectionHandle,
                              SQLINTEGER       Attribute,
                              SQLPOINTER       ValuePtr,
                              SQLINTEGER       BufferLength,
                              SQLINTEGER       *StringLengthPtr);
```

Function arguments

Table 77. SQLGetConnectAttr arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection handle.
SQLINTEGER	*Attribute*	input	*Attribute* to retrieve.
SQLPOINTER	*ValuePtr*	output	A pointer to memory in which to return the current value of the attribute specified by *Attribute*.
SQLINTEGER	*BufferLength*	input	• If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr*. • If *ValuePtr* is a pointer, but not to a string, then *BufferLength* should have the value SQL_IS_POINTER. • If the value in **ValuePtr* is a Unicode string the *BufferLength* argument must be an even number.
SQLINTEGER *	*StringLengthPtr*	output	A pointer to a buffer in which to return the total number of bytes (excluding the null-termination character) available to return in **ValuePtr*. If *ValuePtr* is a null pointer, no length is returned. If the attribute value is a character string, and the number of bytes available to return is greater than BufferLength minus the length of the null-termination character, the data in **ValuePtr* is truncated to *BufferLength* minus the length of the null-termination character and is null-terminated by CLI.

Usage

If *Attribute* specifies an attribute that returns a string, *ValuePtr* must be a pointer to a buffer for the string. The maximum length of the string, including the null termination character, will be *BufferLength* bytes.

Depending on the attribute, an application does not need to establish a connection before calling SQLGetConnectAttr(). However, if SQLGetConnectAttr() is called and the specified attribute does not have a default and has not been set by a prior call to SQLSetConnectAttr(), SQLGetConnectAttr() will return SQL_NO_DATA.

If *Attribute* is SQL_ATTR_TRACE or SQL_ATTR_TRACEFILE, *ConnectionHandle* does not have to be valid, and SQLGetConnectAttr() will not return SQL_ERROR if *ConnectionHandle* is invalid. These attributes apply to all connections. SQLGetConnectAttr() will return SQL_ERROR if another argument is invalid.

While an application can set statement attributes using SQLSetConnectAttr(), an application cannot use SQLGetConnectAttr() to retrieve statement attribute values; it must call SQLGetStmtAttr() to retrieve the setting of statement attributes.

The SQL_ATTR_AUTO_IPD connection attribute can be returned by a call to SQLGetConnectAttr(), but cannot be set by a call to SQLSetConnectAttr().

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 78. SQLGetConnectAttr SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The data returned in **ValuePtr* was truncated to be *BufferLength* minus the length of a null termination character. The length of the untruncated string value is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	An *Attribute* value was specified that required an open connection.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	SQLBrowseConnect() was called for the *ConnectionHandle* and returned SQL_NEED_DATA. This function was called before SQLBrowseConnect() returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* was less than 0.
HY092	Option type out of range.	The value specified for the argument *Attribute* was not valid.
HYC00	Driver not capable.	The value specified for the argument *Attribute* was a valid connection or statement attribute for the version of the CLI driver, but was not supported by the data source.

Restrictions

None.

Example

```
SQLINTEGER autocommit;

/* ... */

/* get the current setting for the AUTOCOMMIT attribute */
cliRC = SQLGetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, &autocommit, 0, NULL);
```

SQLGetConnectOption function (CLI) - Return current setting of a connect option

In ODBC version 3, SQLGetConnectOption() has been deprecated and replaced with SQLGetConnectAttr().

Although this version of CLI continues to support SQLGetConnectOption(), use SQLGetConnectAttr() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLGetConnectOptionW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

The statement:

```
SQLGetConnectOption(hdbc, SQL_ATTR_AUTOCOMMIT, pvAutoCommit);
```

for example, would be rewritten using the new function as:

```
SQLGetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, pvAutoCommit,
                  SQL_IS_POINTER, NULL);
```

SQLGetCursorName function (CLI) - Get cursor name

Returns the cursor name associated with the input statement handle.

If a cursor name was explicitly set by calling SQLSetCursorName(), this name will be returned; otherwise, an implicitly generated name will be returned.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

Unicode Equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLGetCursorNameW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLGetCursorName (
               SQLHSTMT          StatementHandle,  /* hstmt */
               SQLCHAR           *CursorName,      /* szCursor */
               SQLSMALLINT       BufferLength,     /* cbCursorMax */
               SQLSMALLINT       *NameLengthPtr);  /* pcbCursor */
```

Function arguments

Table 79. SQLGetCursorName arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLCHAR *	*CursorName*	output	Cursor name

Table 79. SQLGetCursorName arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*BufferLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store *CursorName*.
SQLSMALLINT *	*NameLengthPtr*	output	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function), excluding the null-termination character, available to return for *CursorName*.

Usage

SQLGetCursorName() will return the cursor name set explicitly with SQLSetCursorName(), or if no name was set, it will return the cursor name internally generated by CLI. If SQLGetCursorName() is called before a statement has been prepared on the input statement handle, an error will result. The internal cursor name is generated on a statement handle the first time dynamic SQL is prepared on the statement handle, not when the handle is allocated.

If a name is set explicitly using SQLSetCursorName(), this name will be returned until the statement is dropped, or until another explicit name is set.

Internally generated cursor names always begin with SQLCUR or SQL_CUR. Cursor names are always 128 SQLCHAR or SQLWCHAR elements or less, and are always unique within a connection.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 80. SQLGetCursorName SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The cursor name returned in *CursorName* was longer than the value in *BufferLength*, and is truncated to *BufferLength* - 1 bytes. The argument *NameLengthPtr* contains the length of the full cursor name available for return. The function returns SQL_SUCCESS_WITH_INFO.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.

Table 80. SQLGetCursorName SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called For the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for the argument *BufferLength* is less than 0.

Restrictions

ODBC generated cursor names start with SQL_CUR, CLI generated cursor names start with SQLCUR, and X/Open CLI generated cursor names begin with either SQLCUR or SQL_CUR.

Example

```
SQLCHAR cursorName[20];

/* ... */

/* get the cursor name of the SELECT statement */
cliRC = SQLGetCursorName(hstmtSelect, cursorName, 20, &cursorLen);
```

SQLGetData function (CLI) - Get data from a column

Retrieves data for a single column in the current row of the result set.

This function is an alternative to SQLBindCol(), which is used to transfer data directly into application variables or LOB locators on each SQLFetch() or SQLFetchScroll() call. An application can either bind LOBs with `SQLBindCol()` or use `SQLGetData()` to retrieve LOBs, but both methods cannot be used together. SQLGetData() can also be used to retrieve large data values in pieces.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

SQLFetch() or `SQLFetchScroll()` must be called before SQLGetData().

After calling SQLGetData() for each column, SQLFetch() or SQLFetchScroll() is called to retrieve the next row.

SQLGetData function (CLI) - Get data from a column

Syntax

```
SQLRETURN   SQLGetData       (
               SQLHSTMT          StatementHandle,     /* hstmt */
               SQLUSMALLINT      ColumnNumber,        /* icol */
               SQLSMALLINT       TargetType,          /* fCType */
               SQLPOINTER        TargetValuePtr,      /* rgbValue */
               SQLLEN            BufferLength,        /* cbValueMax */
               SQLLEN            *StrLen_or_IndPtr);  /* pcbValue */
```

Function arguments

Table 81. SQLGetData arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLUSMALLINT	*ColumnNumber*	input	Column number for which the data retrieval is requested. Result set columns are numbered sequentially from left to right. • Column numbers start at 1 if bookmarks are not used (SQL_ATTR_USE_BOOKMARKS statement attribute set to SQL_UB_OFF). • Column numbers start at 0 if bookmarks are used (the statement attribute set to SQL_UB_ON or SQL_UB_VARIABLE).
SQLSMALLINT	*TargetType*	input	The C data type of the column identifier by *ColumnNumber*. The following types are supported: • SQL_C_BINARY • SQL_C_BIT • SQL_C_BLOB_LOCATOR • SQL_C_CHAR • SQL_C_CLOB_LOCATOR • SQL_C_DBCHAR • SQL_C_DBCLOB_LOCATOR • SQL_C_DECIMAL_IBM • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_NUMERIC [a] • SQL_C_SBIGINT • SQL_C_SHORT • SQL_C_TYPE_DATE • SQL_C_TYPE_TIME • SQL_C_TYPE_TIMESTAMP • SQL_C_TYPE_TIMESTAMP_EXT • SQL_C_TINYINT • SQL_C_UBIGINT • SQL_C_UTINYINT • SQL_C_WCHAR Specifying SQL_ARD_TYPE results in the data being converted to the data type specified in the SQL_DESC_CONCISE_TYPE field of the ARD. Specifying SQL_C_DEFAULT results in the data being converted to its default C data type.
SQLPOINTER	*TargetValuePtr*	output	Pointer to buffer where the retrieved column data is to be stored.
SQLLEN	*BufferLength*	input	Maximum size of the buffer pointed to by *TargetValuePtr*. This value is ignored when the driver returns fixed-length data.

Table 81. SQLGetData arguments (continued)

Data type	Argument	Use	Description
SQLLEN *	*StrLen_or_IndPtr*	output	Pointer to value which indicates the number of bytes CLI has available to return in the *TargetValuePtr* buffer. If the data is being retrieved in pieces, this contains the number of bytes still remaining. The value is SQL_NULL_DATA if the data value of the column is null. If this pointer is NULL and SQLFetch() has obtained a column containing null data, then this function will fail because it has no means of reporting this. If SQLFetch() has fetched a column containing binary data, then the pointer to *StrLen_or_IndPtr* must not be NULL or this function will fail because it has no other means of informing the application about the length of the data retrieved in the *TargetValuePtr* buffer.

Note: CLI will provide some performance enhancement if *TargetValuePtr* is placed consecutively in memory after *StrLen_or_IndPtr*

Usage

Different DB2 data sources have different restrictions on how `SQLGetData()` can be used. For an application to be sure about the functional capabilities of this function, it should call `SQLGetInfo()` with any of the following SQL_GETDATA_EXTENSIONS options:

- SQL_GD_ANY_COLUMN: If this option is returned, `SQLGetData()` can be called for any unbound column, including those before the last bound column. All DB2 data sources support this feature.
- SQL_GD_ANY_ORDER: If this option is returned, `SQLGetData()` can be called for unbound columns in any order. All DB2 data sources support this feature.
- SQL_GD_BLOCK: If this option if returned by `SQLGetInfo()` for the SQL_GETDATA_EXTENSIONS *InfoType* argument, then the driver will support calls to `SQLGetData()` when the rowset size is greater than 1. The application can also call `SQLSetPos()` with the SQL_POSITION option to position the cursor on the correct row before calling `SQLGetData()`. At least DB2 for UNIX and Windows data sources support this feature.
- SQL_GD_BOUND: If this option is returned, `SQLGetData()` can be called for bound columns as well as unbound columns. DB2 Database for Linux, UNIX, and Windows does not currently support this feature.

SQLGetData() can also be used to retrieve long columns if the C data type (*TargetType*) is SQL_C_CHAR, SQL_C_BINARY, SQL_C_DBCHAR, SQL_C_WCHAR, or if *TargetType* is SQL_C_DEFAULT and the column type denotes a binary or character string.

Upon each SQLGetData() call, if the data available for return is greater than or equal to *BufferLength*, truncation occurs. Truncation is indicated by a function return code of SQL_SUCCESS_WITH_INFO coupled with a SQLSTATE denoting data truncation. The application can call SQLGetData() again, with the same ColumnNumber value, to get subsequent data from the same unbound column starting at the point of truncation. To obtain the entire column, the application

repeats such calls until the function returns SQL_SUCCESS. The next call to SQLGetData() returns SQL_NO_DATA_FOUND.

When the application calls the function SQLGetData() to retrieve the actual LOB data it will, by default, make one request to the server and will store the entire LOB in memory provided *BufferLength* is large enough. If *BufferLength* is not large enough to hold the requested LOB data, it will be retrieved in pieces.

Although SQLGetData() can be used for the sequential retrieval of LOB column data, use the CLI LOB functions if only a portion of the LOB data or a few sections of the LOB column data are needed:

1. Bind the column to a LOB locator.
2. Fetch the row.
3. Use the locator in a SQLGetSubString() call, to retrieve the data in pieces (SQLGetLength() and SQLGetPosition() might also be required in order to determine the values of some of the arguments).
4. Repeat step 2.

Truncation is also affected by the SQL_ATTR_MAX_LENGTH statement attribute. The application can specify that truncation is not to be reported by calling SQLSetStmtAttr() with SQL_ATTR_MAX_LENGTH and a value for the maximum length to return for any one column, and by allocating a *TargetValuePtr* buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, SQL_SUCCESS will be returned and the maximum length, not the actual length will be returned in *StrLen_or_IndPtr*.

To discard the column data part way through the retrieval, the application can call SQLGetData() with *ColumnNumber* set to the next column position of interest. To discard data that has not been retrieved for the entire row, the application should call SQLFetch() to advance the cursor to the next row; or, if it does not want any more data from the result set, the application can close the cursor by calling `SQLCloseCursor()` or `SQLFreeStmt()` with the SQL_CLOSE or SQL_DROP option.

The *TargetType* input argument determines the type of data conversion (if any) needed before the column data is placed into the storage area pointed to by *TargetValuePtr*.

For SQL graphic column data:

- The length of the *TargetValuePtr* buffer (*BufferLength*) should be a multiple of 2. The application can determine the SQL data type of the column by first calling SQLDescribeCol() or SQLColAttribute().
- The pointer to *StrLen_or_IndPtr* must not be NULL since CLI will be storing the number of octets stored in *TargetValuePtr*.
- If the data is to be retrieved in piecewise fashion, CLI will attempt to fill *TargetValuePtr* to the nearest multiple of two octets that is still less than or equal to *BufferLength*. This means if *BufferLength* is not a multiple of two, the last byte in that buffer will be untouched; CLI will not split a double-byte character.

The content returned in *TargetValuePtr* is always null-terminated unless the column data to be retrieved is binary, or if the SQL data type of the column is graphic (DBCS) and the C buffer type is SQL_C_CHAR. If the application is retrieving the data in multiple chunks, it should make the proper adjustments (for example, strip off the null-terminator before concatenating the pieces back together assuming the null termination environment attribute is in effect).

Truncation of numeric data types is reported as a warning if the truncation involves digits to the right of the decimal point. If truncation occurs to the left of the decimal point, an error is returned (refer to the diagnostics section).

With the exception of scrollable cursors, applications that use SQLFetchScroll() to retrieve data should call SQLGetData() only when the rowset size is 1 (equivalent to issuing SQLFetch()). SQLGetData() can only retrieve column data for a row where the cursor is currently positioned.

Using SQLGetData() with Scrollable Cursors

SQLGetData() can also be used with scrollable cursors. You can save a pointer to any row in the result set with a bookmark. The application can then use that bookmark as a relative position to retrieve a rowset of information.

Once you have positioned the cursor to a row in a rowset using SQLSetPos(), you can obtain the bookmark value from column 0 using SQLGetData(). In most cases you will not want to bind column 0 and retrieve the bookmark value for every row, but use SQLGetData() to retrieve the bookmark value for the specific row you require.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND
- SQL_NO_TOTAL

SQL_NO_DATA_FOUND is returned when the preceding SQLGetData() call has retrieved all of the data for this column.

SQL_SUCCESS is returned if a zero-length string is retrieved by SQLGetData(). If this is the case, StrLen_or_IndPtr will contain 0, and TargetValuePtr will contain a null terminator.

SQL_NO_TOTAL is returned as the length when truncation occurs if the CLI configuration keyword StreamGetData is set to 1 and CLI cannot determine the number of bytes still available to return in the output buffer.

If the preceding call to SQLFetch() failed, SQLGetData() should not be called since the result is undefined.

Diagnostics

Table 82. SQLGetData SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	Data returned for the specified column (ColumnNumber) was truncated. String or numeric values are right truncated. SQL_SUCCESS_WITH_INFO is returned.
07006	Invalid conversion.	The data value cannot be converted to the C data type specified by the argument *TargetType*. The function has been called before for the same *ColumnNumber* value but with a different *TargetType* value.

Table 82. SQLGetData SQLSTATEs (continued)

SQLSTATE	Description	Explanation
07009	Invalid descriptor index.	The value specified for *ColumnNumber* was equal to 0, and the SQL_ATTR_USE_BOOKMARKS statement attribute was SQL_UB_OFF. The value specified for the argument *ColumnNumber* was greater than the number of columns in the result set.
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument StrLen_or_IndPtr was a null pointer and the value of the column is null. There is no means to report SQL_NULL_DATA.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for the column would have caused the whole part of the number to be truncated.
22005	Error in assignment.	A returned value was incompatible with the data type denoted by the argument *TargetType*.
22007	Invalid datetime format.	Conversion from character a string to a datetime format was indicated, but an invalid string representation or value was specified, or the value was an invalid date.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
24000	Invalid cursor state.	The previous SQLFetch() resulted in SQL_ERROR or SQL_NO_DATA found; as a result, the cursor is not positioned on a row.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY003	Program type out of range.	*TargetType* was not a valid data type or SQL_C_DEFAULT.
HY010	Function sequence error.	The specified *StatementHandle* was not in a cursor positioned state. The function was called without first calling SQLFetch(). The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called For the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY011	Operation invalid at this time.	Calls to `SQLGetData()` for previously accessed LOB columns are not allowed. Refer to "AllowGetDataLOBReaccess CLI/ODBC configuration keyword" on page 327 for more information.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Table 82. SQLGetData SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY090	Invalid string or buffer length.	The value of the argument *BufferLength* is less than 0 and the argument *TargetType* is SQL_C_CHAR, SQL_C_BINARY, SQL_C_DBCHAR or (SQL_C_DEFAULT and the default type is one of SQL_C_CHAR, SQL_C_BINARY, or SQL_C_DBCHAR).
HYC00	Driver not capable.	The SQL data type for the specified data type is recognized but not supported by CLI. The requested conversion from the SQL data type to the application data *TargetType* cannot be performed by CLI or the data source. The column was bound using SQLBindFileToCol().
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

Example

```
/* use SQLGetData to get the results */
/* get data from column 1 */
cliRC = SQLGetData(hstmt,
                   1,
                   SQL_C_SHORT,
                   &deptnumb.val,
                   0,
                   &deptnumb.ind);
STMT_HANDLE_CHECK(hstmt, hdbc, cliRC);

/* get data from column 2 */
cliRC = SQLGetData(hstmt,
                   2,
                   SQL_C_CHAR,
                   location.val,
                   15,
                   &location.ind);
```

SQLGetDescField function (CLI) - Get single field settings of descriptor record

Returns the current settings of a single field of a descriptor record.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLGetDescFieldW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

SQLGetDescField function (CLI) - Get single field settings of descriptor record

Syntax

```
SQLRETURN   SQLGetDescField (
               SQLHDESC          DescriptorHandle,
               SQLSMALLINT       RecNumber,
               SQLSMALLINT       FieldIdentifier,
               SQLPOINTER        ValuePtr,           /* Value */
               SQLINTEGER        BufferLength,
               SQLINTEGER        *StringLengthPtr);  /* *StringLength */
```

Function arguments

Table 83. SQLGetDescField arguments

Data type	Argument	Use	Description
SQLHDESC	*DescriptorHandle*	input	Descriptor handle.
SQLSMALLINT	*RecNumber*	input	Indicates the descriptor record from which the application seeks information. Descriptor records are numbered from 0, with record number 0 being the bookmark record. If the *FieldIdentifier* argument indicates a field of the descriptor header record, *RecNumber* must be 0. If *RecNumber* is less than SQL_DESC_COUNT, but the row does not contain data for a column or parameter, a call to SQLGetDescField() will return the default values of the fields.
SQLSMALLINT	*FieldIdentifier*	input	Indicates the field of the descriptor whose value is to be returned.
SQLPOINTER	*ValuePtr*	output	Pointer to a buffer in which to return the descriptor information. The data type depends on the value of *FieldIdentifier*.
SQLINTEGER	*BufferLength*	input	• If *ValuePtr* points to a character string, this argument should be the length of **ValuePtr*. • If *ValuePtr* is a pointer, but not to a string, then *BufferLength* should have the value SQL_IS_POINTER. • If the value in **ValuePtr* is of a Unicode data type the *BufferLength* argument must be an even number.
SQLSMALLINT *	*StringLengthPtr*	output	Pointer to the total number of bytes (excluding the number of bytes required for the null termination character) available to return in **ValuePtr*.

Usage

An application can call SQLGetDescField() to return the value of a single field of a descriptor record. A call to SQLGetDescField() can return the setting of any field in any descriptor type, including header fields, record fields, and bookmark fields. An application can obtain the settings of multiple fields in the same or different descriptors, in arbitrary order, by making repeated calls to SQLGetDescField(). SQLGetDescField() can also be called to return CLI defined descriptor fields.

For performance reasons, an application should not call SQLGetDescField() for an IRD before executing a statement. Calling SQLGetDescField() in this case causes the CLI driver to describe the statement, resulting in an extra network flow. When deferred prepare is on and SQLGetDescField() is called, you lose the benefit of deferred prepare because the statement must be prepared at the server to obtain describe information.

The settings of multiple fields that describe the name, data type, and storage of column or parameter data can also be retrieved in a single call to `SQLGetDescRec()`. `SQLGetStmtAttr()` can be called to return the value of a single field in the descriptor header that has an associated statement attribute.

When an application calls SQLGetDescField() to retrieve the value of a field that is undefined for a particular descriptor type, the function returns SQLSTATE HY091 (Invalid descriptor field identifier). When an application calls SQLGetDescField() to retrieve the value of a field that is defined for a particular descriptor type, but has no default value and has not been set yet, the function returns SQL_SUCCESS but the value returned for the field is undefined. Refer to the list of initialization values of descriptor fields for any default values which may exist.

The SQL_DESC_ALLOC_TYPE header field is available as read-only. This field is defined for all types of descriptors.

Each of these fields is defined either for the IRD only, or for both the IRD and the IPD.

```
SQL_DESC_AUTO_UNIQUE_VALUE    SQL_DESC_LITERAL_SUFFIX
SQL_DESC_BASE_COLUMN_NAME     SQL_DESC_LOCAL_TYPE_NAME
SQL_DESC_CASE_SENSITIVE       SQL_DESC_SCHEMA_NAME
SQL_DESC_CATALOG_NAME         SQL_DESC_SEARCHABLE
SQL_DESC_DISPLAY_SIZE         SQL_DESC_TABLE_NAME
SQL_DESC_FIXED_PREC_SCALE     SQL_DESC_TYPE_NAME
SQL_DESC_LABEL                SQL_DESC_UNSIGNED
SQL_DESC_LITERAL_PREFIX       SQL_DESC_UPDATABLE
```

Refer to the list of descriptor *FieldIdentifier* values for more information about the listed fields.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_NO_DATA
- SQL_INVALID_HANDLE

SQL_NO_DATA is returned if *RecNumber* is greater than the number of descriptor records.

SQL_NO_DATA is returned if *DescriptorHandle* is an IRD handle and the statement is in the prepared or executed state, but there was no open cursor associated with it.

Diagnostics

Table 84. SQLGetDescField SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The buffer **ValuePtr* was not large enough to return the entire descriptor field, so the field was truncated. The length of the untruncated descriptor field is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)

Table 84. SQLGetDescField SQLSTATEs (continued)

SQLSTATE	Description	Explanation
07009	Invalid descriptor index.	The value specified for the *RecNumber* argument was less than 1, the SQL_ATTR_USE_BOOKMARKS statement attribute was SQL_UB_OFF, and the field was not a header field or a CLI defined field. The *FieldIdentifier* argument was a record field, and the *RecNumber* argument was 0. The *RecNumber* argument was less than 0, and the field was not a header field or a CLI defined field.
08S01	Communication link failure.	The communication link between CLI and the data source to which it was connected failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY007	Associated statement is not prepared.	*DescriptorHandle* was associated with an IRD, and the associated statement handle was not in the prepared or executed state.
HY010	Function sequence error.	*DescriptorHandle* was associated with a StatementHandle for which an asynchronously executing function (not this one) was called and was still executing when this function was called. *DescriptorHandle* was associated with a *StatementHandle* for which `SQLExecute()` or `SQLExecDirect()` was called and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY021	Inconsistent descriptor information.	The descriptor information checked during a consistency check was not consistent.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.
HY091	Invalid descriptor field identifier.	*FieldIdentifier* was undefined for the *DescriptorHandle*. The value specified for the *RecNumber* argument was greater than the value in the SQL_DESC_COUNT field.

Restrictions

None.

Example

```
/* see how the field SQL_DESC_PARAMETER_TYPE is set */
cliRC = SQLGetDescField(hIPD,
                        1, /* look at the parameter */
                        SQL_DESC_PARAMETER_TYPE,
                        &descFieldParameterType, /* result */
                        SQL_IS_SMALLINT,
```

```
                        NULL);

/* ... */

  /* see how the descriptor record field SQL_DESC_TYPE_NAME is set */
  rc = SQLGetDescField(hIRD,
                       (SQLSMALLINT)colCount,
                       SQL_DESC_TYPE_NAME, /* record field */
                       descFieldTypeName, /* result */
                       25,
                       NULL);
  /* ... */

  /* see how the descriptor record field SQL_DESC_LABEL is set */
  rc = SQLGetDescField(hIRD,
                       (SQLSMALLINT)colCount,
                       SQL_DESC_LABEL, /* record field */
                       descFieldLabel, /* result */
                       25,
                       NULL);
```

SQLGetDescRec function (CLI) - Get multiple field settings of descriptor record

Returns the current settings of multiple fields of a descriptor record.

The fields returned describe the name, data type, and storage of column or parameter data.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLGetDescRecW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLGetDescRec (
               SQLHDESC          DescriptorHandle,   /* hDesc */
               SQLSMALLINT       RecNumber,
               SQLCHAR           *Name,
               SQLSMALLINT       BufferLength,
               SQLSMALLINT       *StringLengthPtr,   /* *StringLength */
               SQLSMALLINT       *TypePtr,           /* *Type */
               SQLSMALLINT       *SubTypePtr,        /* *SubType */
               SQLLEN            *LengthPtr,         /* *Length */
               SQLSMALLINT       *PrecisionPtr,      /* *Precision */
               SQLSMALLINT       *ScalePtr,          /* *Scale */
               SQLSMALLINT       *NullablePtr);      /* *Nullable */
```

Function arguments

Table 85. SQLGetDescRec arguments

Data type	Argument	Use	Description
SQLHDESC	*DescriptorHandle*	input	Descriptor handle.

Table 85. SQLGetDescRec arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*RecNumber*	input	Indicates the descriptor record from which the application seeks information. Descriptor records are numbered from 0, with record number 0 being the bookmark record. The *RecNumber* argument must be less than or equal to the value of SQL_DESC_COUNT. If *RecNumber* is less than SQL_DESC_COUNT, but the row does not contain data for a column or parameter, a call to SQLGetDescRec() will return the default values of the fields.
SQLCHAR *	*Name*	output	A pointer to a buffer in which to return the SQL_DESC_NAME field for the descriptor record.
SQLINTEGER	*BufferLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the **Name* buffer.
SQLSMALLINT *	*StringLengthPtr*	output	A pointer to a buffer in which to return the number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) available to return in the *Name* buffer, excluding the null-termination character. If the number of SQLCHAR or SQLWCHAR elements was greater than or equal to *BufferLength*, the data in **Name* is truncated to *BufferLength* minus the length of a null-termination character, and is null terminated by CLI.
SQLSMALLINT *	*TypePtr*	output	A pointer to a buffer in which to return the value of the SQL_DESC_TYPE field for the descriptor record.
SQLSMALLINT *	*SubTypePtr*	output	For records whose type is SQL_DATETIME, this is a pointer to a buffer in which to return the value of the SQL_DESC_DATETIME_INTERVAL_CODE field.
SQLLEN *	*LengthPtr*	output	A pointer to a buffer in which to return the value of the SQL_DESC_OCTET_LENGTH field for the descriptor record.
SQLSMALLINT *	*PrecisionPtr*	output	A pointer to a buffer in which to return the value of the SQL_DESC_PRECISION field for the descriptor record.
SQLSMALLINT *	*ScalePtr*	output	A pointer to a buffer in which to return the value of the SQL_DESC_SCALE field for the descriptor record.
SQLSMALLINT *	*NullablePtr*	output	A pointer to a buffer in which to return the value of the SQL_DESC_NULLABLE field for the descriptor record.

Usage

An application can call SQLGetDescRec() to retrieve the values of the following fields for a single column or parameter:

- SQL_DESC_NAME
- SQL_DESC_TYPE
- SQL_DESC_DATETIME_INTERVAL_CODE (for records whose type is SQL_DATETIME)
- SQL_DESC_OCTET_LENGTH

- SQL_DESC_PRECISION
- SQL_DESC_SCALE
- SQL_DESC_NULLABLE

SQLGetDescRec() does not retrieve the values for header fields.

An application can inhibit the return of a field's setting by setting the argument corresponding to the field to a null pointer. When an application calls SQLGetDescRec() to retrieve the value of a field that is undefined for a particular descriptor type, the function returns SQL_SUCCESS but the value returned for the field is undefined. For example, calling SQLGetDescRec() for the SQL_DESC_NAME or SQL_DESC_NULLABLE field of an APD or ARD will return SQL_SUCCESS but an undefined value for the field.

When an application calls SQLGetDescRec() to retrieve the value of a field that is defined for a particular descriptor type, but has no default value and has not been set yet, the function returns SQL_SUCCESS but the value returned for the field is undefined.

The values of fields can also be retrieved individually by a call to SQLGetDescField().

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_NO_DATA
- SQL_INVALID_HANDLE

SQL_NO_DATA is returned if *RecNumber* is greater than the number of descriptor records.

SQL_NO_DATA is returned if *DescriptorHandle* is an IRD handle and the statement in the prepared or executed state, but there was no open cursor associated with it.

Diagnostics

Table 86. SQLGetDescRec SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The buffer **Name* was not large enough to return the entire descriptor field, so the field was truncated. The length of the untruncated descriptor field is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)
07009	Invalid descriptor index.	The *RecNumber* argument was set to 0 and the *DescriptorHandle* argument was an IPD handle. The *RecNumber* argument was set to 0, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF. The *RecNumber* argument was less than 0.
08S01	Communication link failure.	The communication link between CLI and the data source to which it was connected failed before the function completed processing.

Table 86. SQLGetDescRec SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY007	Associated statement is not prepared.	*DescriptorHandle* was associated with an IRD, and the associated statement handle was not in the prepared or executed state.
HY010	Function sequence error.	*DescriptorHandle* was associated with a *StatementHandle* for which an asynchronously executing function (not this one) was called and was still executing when this function was called. *DescriptorHandle* was associated with a *StatementHandle* for which SQLExecute() or SQLExecDirect() was called and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Restrictions

None.

Example

```
/* get multiple field settings of descriptor record */
rc = SQLGetDescRec(hIRD,
                   i,
                   colname,
                   sizeof(colname),
                   &namelen,
                   &type,
                   &subtype,
                   &width,
                   &precision,
                   &scale,
                   &nullable);

/* ... */

/* get the record/column value after setting */
rc = SQLGetDescRec(hARD,
                   i,
                   colname,
                   sizeof(colname),
                   &namelen,
                   &type,
                   &subtype,
                   &width,
                   &precision,
                   &scale,
                   &nullable);
```

SQLGetDiagField function (CLI) - Get a field of diagnostic data

Returns the current value of a field of a diagnostic data structure, associated with a specific handle, that contains error, warning, and status information.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLGetDiagFieldW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLGetDiagField (
               SQLSMALLINT       HandleType,         /* fHandleType */
               SQLHANDLE         Handle,             /* hHandle */
               SQLSMALLINT       RecNumber,          /* iRecNumber */
               SQLSMALLINT       DiagIdentifier,     /* fDiagIdentifier */
               SQLPOINTER        DiagInfoPtr,        /* pDiagInfo */
               SQLSMALLINT       BufferLength,       /* cbDiagInfoMax */
               SQLSMALLINT       *StringLengthPtr);  /* *pcgDiagInfo */
```

Function arguments

Table 87. SQLGetDiagField arguments

Data type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	A handle type identifier that describes the type of handle for which diagnostics are required. The handle type identifier include: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC
SQLHANDLE	*Handle*	input	A handle for the diagnostic data structure, of the type indicated by *HandleType*.
SQLSMALLINT	*RecNumber*	input	Indicates the status record from which the application seeks information. Status records are numbered from 1. If the *DiagIdentifier* argument indicates any field of the diagnostics header record, *RecNumber* must be 0. If not, it should be greater than 0.
SQLSMALLINT	*DiagIdentifier*	input	Indicates the field of the diagnostic data structure whose value is to be returned. For more information, see DiagIdentifier argument.
SQLPOINTER	*DiagInfoPtr*	output	Pointer to a buffer in which to return the diagnostic information. The data type depends on the value of *DiagIdentifier*.

Table 87. SQLGetDiagField arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER	*BufferLength*	input	If *DiagIdentifier* is ODBC-defined diagnostic: • If *DiagInfoPtr* points to a character string or binary buffer, *BufferLength* should be the length of **DiagInfoPtr*. • If **DiagInfoPtr* is an integer, *BufferLength* is ignored. • If **DiagInfoPtr* is a Unicode string, *BufferLength* must be an even number. If *DiagIdentifier* is a CLI diagnostic: • If **DiagInfoPtr* is a pointer to a character string, *BufferLength* is the number of bytes needed to store the string, or SQL_NTS. • If **DiagInfoPtr* is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in *BufferLength*. This places a negative value in *BufferLength*. • If **DiagInfoPtr* is a pointer to a value other than a character string or binary string, then *BufferLength* should have the value SQL_IS_POINTER. • If **DiagInfoPtr* contains a fixed-length data type, then *BufferLength* is SQL_IS_INTEGER, SQL_IS_UINTEGER, SQL_IS_SMALLINT, or SQL_IS_USMALLINT, as appropriate.
SQLSMALLINT *	*StringLengthPtr*	output	Pointer to a buffer in which to return the total number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function), excluding the number of bytes required for the null-termination character, available to return in **DiagInfoPtr*, for character data. If the number of bytes available to return is greater than *BufferLength*, then the text in **DiagInfoPtr* is truncated to *BufferLength* minus the length of a null-termination character. This argument is ignored for non-character data.

Usage

An application typically calls SQLGetDiagField() to accomplish one of three goals:

1. To obtain specific error or warning information when a function call has returned the SQL_ERROR or SQL_SUCCESS_WITH_INFO (or SQL_NEED_DATA for the `SQLBrowseConnect()` function) return codes.
2. To find out the number of rows in the data source that were affected when insert, delete, or update operations were performed with a call to `SQLExecute()`, `SQLExecDirect()`, `SQLBulkOperations()`, or `SQLSetPos()` (from the SQL_DIAG_ROW_COUNT header field), or to find out the number of rows that exist in the current open static scrollable cursor (from the SQL_DIAG_CURSOR_ROW_COUNT header field).
3. To determine which function was executed by a call to `SQLExecDirect()` or `SQLExecute()` (from the SQL_DIAG_DYNAMIC_FUNCTION and SQL_DIAG_DYNAMIC_FUNCTION_CODE header fields).

Any CLI function can post zero or more errors each time it is called, so an application can call SQLGetDiagField() after any function call. SQLGetDiagField() retrieves only the diagnostic information most recently associated with the diagnostic data structure specified in the *Handle* argument. If the application calls another function, any diagnostic information from a previous call with the same handle is lost.

An application can scan all diagnostic records by incrementing *RecNumber*, as long as SQLGetDiagField() returns SQL_SUCCESS. The number of status records is indicated in the SQL_DIAG_NUMBER header field. Calls to SQLGetDiagField() are non-destructive as far as the header and status records are concerned. The application can call SQLGetDiagField() again at a later time to retrieve a field from a record, as long as another function other than SQLGetDiagField(), `SQLGetDiagRec()`, or `SQLError()` has not been called in the interim, which would post records on the same handle.

An application can call SQLGetDiagField() to return any diagnostic field at any time, with the exception of SQL_DIAG_ROW_COUNT, which will return SQL_ERROR if *Handle* was not a statement handle on which an SQL statement had been executed. If any other diagnostic field is undefined, the call to SQLGetDiagField() will return SQL_SUCCESS (provided no other error is encountered), and an undefined value is returned for the field.

HandleType argument

Each handle type can have diagnostic information associated with it. The *HandleType* argument denotes the handle type of *Handle*.

Some header and record fields cannot be returned for all types of handles: environment, connection, statement, and descriptor. Those handles for which a field is not applicable are indicated in the Header Field and Record Fields sections.

No CLI-specific header diagnostic field should be associated with an environment handle.

DiagIdentifier argument

This argument indicates the identifier of the field required from the diagnostic data structure. If *RecNumber* is greater than or equal to 1, the data in the field describes the diagnostic information returned by a function. If *RecNumber* is 0, the field is in the header of the diagnostic data structure, so it contains data pertaining to the function call that returned the diagnostic information, not the specific information. Refer to the list of header and record fields for the *DiagIdentifier* argument for further information.

Sequence of status records

Status records are placed in a sequence based upon row number and the type of the diagnostic.

If there are two or more status records, the sequence of the records is determined first by row number. The following rules apply to determining the sequence of errors by row:

- Records that do not correspond to any row appear in front of records that correspond to a particular row, since SQL_NO_ROW_NUMBER is defined to be -1.

- Records for which the row number is unknown appear in front of all other records, since SQL_ROW_NUMBER_UNKNOWN is defined to be -2.
- For all records that pertain to specific rows, records are sorted by the value in the SQL_DIAG_ROW_NUMBER field. All errors and warnings of the first row affected are listed, then all errors and warnings of the next row affected, and so on.

Within each row, or for all those records that do not correspond to a row or for which the row number is unknown, the first record listed is determined using a set of sorting rules. After the first record, the order of the other records affecting a row is undefined. An application cannot assume that errors precede warnings after the first record. Applications should scan the entire diagnostic data structure to obtain complete information about an unsuccessful call to a function.

The following rules are followed to determine the first record within a row. The record with the highest rank is the first record.

- **Errors**. Status records that describe errors have the highest rank. The following rules are followed to sort errors:
  - Records that indicate a transaction failure or possible transaction failure outrank all other records.
  - If two or more records describe the same error condition, then SQLSTATEs defined by the X/Open CLI specification (classes 03 through HZ) outrank ODBC- and driver-defined SQLSTATEs.
- **Implementation-defined No Data values**. Status records that describe CLI No Data values (class 02) have the second highest rank.
- **Warnings**. Status records that describe warnings (class 01) have the lowest rank. If two or more records describe the same warning condition, then warning SQLSTATEs defined by the X/Open CLI specification outrank ODBC- and driver-defined SQLSTATEs.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA

Diagnostics

SQLGetDiagField() does not post error values for itself. It uses the following return values to report the outcome of its own execution:

- SQL_SUCCESS: The function successfully returned diagnostic information.
- SQL_SUCCESS_WITH_INFO: **DiagInfoPtr* was too small to hold the requested diagnostic field so the data in the diagnostic field was truncated. To determine that a truncation occurred, the application must compare *BufferLength* to the actual number of bytes available, which is written to **StringLengthPtr*.
- SQL_INVALID_HANDLE: The handle indicated by *HandleType* and *Handle* was not a valid handle.
- SQL_ERROR: Possible causes are:
  - The *DiagIdentifier* argument was not one of the valid values.
  - The *DiagIdentifier* argument was SQL_DIAG_CURSOR_ROW_COUNT, SQL_DIAG_DYNAMIC_FUNCTION,

SQL_DIAG_DYNAMIC_FUNCTION_CODE, or SQL_DIAG_ROW_COUNT, but *Handle* was not a statement handle.
  - The *RecNumber* argument was negative or 0 when *DiagIdentifier* indicated a field from a diagnostic record. *RecNumber* is ignored for header fields.
  - The value requested was a character string and *BufferLength* was less than zero.
- SQL_NO_DATA: *RecNumber* was greater than the number of diagnostic records that existed for the handle specified in *Handle*. The function also returns SQL_NO_DATA for any positive *RecNumber* if there are no diagnostic records for *Handle*.

Restrictions

None.

SQLGetDiagRec function (CLI) - Get multiple fields settings of diagnostic record

Returns the current values of multiple fields of a diagnostic record that contains error, warning, and status information.

Unlike SQLGetDiagField(), which returns one diagnostic field per call, SQLGetDiagRec() returns several commonly used fields of a diagnostic record: the SQLSTATE, native error code, and error message text.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLGetDiagRecW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLGetDiagRec (
               SQLSMALLINT       HandleType,       /* fHandleType */
               SQLHANDLE         Handle,           /* hHandle */
               SQLSMALLINT       RecNumber,        /* iRecNumber */
               SQLCHAR           *SQLState,        /* *pszSqlState */
               SQLINTEGER        *NativeErrorPtr,  /* *pfNativeError */
               SQLCHAR           *MessageText,     /* *pszErrorMsg */
               SQLSMALLINT       BufferLength,     /* cbErrorMsgMax */
               SQLSMALLINT       *TextLengthPtr);  /* *pcbErrorMsg */
```

Function arguments

Table 88. SQLGetDiagRec arguments

Data type	Argument	Use	Description
SQLSMALLINT	*HandleType*	input	A handle type identifier that describes the type of handle for which diagnostics are desired. The handle type identifier include: • SQL_HANDLE_ENV • SQL_HANDLE_DBC • SQL_HANDLE_STMT • SQL_HANDLE_DESC
SQLHANDLE	*Handle*	input	A handle for the diagnostic data structure, of the type indicated by *HandleType*.
SQLSMALLINT	*RecNumber*	input	Indicates the status record from which the application seeks information. Status records are numbered from 1.
SQLCHAR *	*SQLState*	output	Pointer to a buffer in which to return 5 characters plus a NULL terminator for the SQLSTATE code pertaining to the diagnostic record *RecNumber*. The first two characters indicate the class; the next three indicate the subclass.
SQLINTEGER *	*NativeErrorPtr*	output	Pointer to a buffer in which to return the native error code, specific to the data source.
SQLCHAR *	*MessageText*	output	Pointer to a buffer in which to return the error message text. The fields returned by SQLGetDiagRec() are contained in a text string.
SQLINTEGER	*BufferLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the *MessageText* buffer.
SQLSMALLINT *	*TextLengthPtr*	output	Pointer to a buffer in which to return the total number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function), excluding the null-termination character, available to return in **MessageText*. If the number of SQLCHAR or SQLWCHAR elements available to return is greater than *BufferLength*, then the error message text in **MessageText* is truncated to *BufferLength* minus the length of a null-termination character.

Usage

An application typically calls SQLGetDiagRec() when a previous call to a CLI function has returned anything other than SQL_SUCCESS. However, any function can post zero or more errors each time it is called, so an application can call SQLGetDiagRec() after any function call. An application can call SQLGetDiagRec() multiple times to return some or all of the records in the diagnostic data structure.

SQLGetDiagRec() returns a character string containing the following fields of the diagnostic data structure record:

SQL_DIAG_MESSAGE_TEXT (return type CHAR *)
An informational message on the error or warning.

SQL_DIAG_NATIVE (return type SQLINTEGER)
A driver/data-source-specific native error code. If there is no native error code, the driver returns 0.

SQL_DIAG_SQLSTATE (return type CHAR *)
A five-character SQLSTATE diagnostic code.

SQLGetDiagRec() cannot be used to return fields from the header of the diagnostic data structure (the *RecNumber* argument must be greater than 0). The application should call SQLGetDiagField() for this purpose.

SQLGetDiagRec() retrieves only the diagnostic information most recently associated with the handle specified in the *Handle* argument. If the application calls another function, except SQLGetDiagRec() or SQLGetDiagField(), any diagnostic information from the previous calls on the same handle is lost.

An application can scan all diagnostic records by looping, incrementing *RecNumber*, as long as SQLGetDiagRec() returns SQL_SUCCESS. Calls to SQLGetDiagRec() are non-destructive to the header and record fields. The application can call SQLGetDiagRec() again at a later time to retrieve a field from a record, as long as no other function, except SQLGetDiagRec() or SQLGetDiagField(), has been called in the interim. The application can call SQLGetDiagField() to retrieve the value of the SQL_DIAG_NUMBER field, which is the total number of diagnostic records available. SQLGetDiagRec() should then be called that many times.

HandleType argument

Each handle type can have diagnostic information associated with it. The *HandleType* argument denotes the handle type of *Handle*.

Some header and record fields cannot be returned for all types of handles: environment, connection, statement, and descriptor. Those handles for which a field is not applicable are indicated in the list of header and record fields for the *DiagIdentifier* argument.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

SQLGetDiagRec() does not post error values for itself. It uses the following return values to report the outcome of its own execution:

- SQL_SUCCESS: The function successfully returned diagnostic information.
- SQL_SUCCESS_WITH_INFO: The **MessageText* buffer was too small to hold the requested diagnostic message. No diagnostic records were generated. To determine that a truncation occurred, the application must compare *BufferLength* to the actual number of bytes available, which is written to **StringLengthPtr*.
- SQL_INVALID_HANDLE: The handle indicated by *HandleType* and *Handle* was not a valid handle.
- SQL_ERROR: Possible causes are:
  - *RecNumber* was negative or 0.
  - *BufferLength* was less than zero.
- SQL_NO_DATA: *RecNumber* was greater than the number of diagnostic records that existed for the handle specified in *Handle*. The function also returns SQL_NO_DATA for any positive *RecNumber* if there are no diagnostic records for *Handle*.

Example

```
/* get multiple fields settings of diagnostic record */
SQLGetDiagRec(SQL_HANDLE_STMT,
              hstmt,
              1,
              sqlstate,
              &sqlcode,
              message,
              200,
              &length);
```

SQLGetEnvAttr function (CLI) - Retrieve current environment attribute value

Returns the current setting for the specified environment attribute.

These options are set using the SQLSetEnvAttr() function.

Specification:

- **CLI** 2.1
- **ISO CLI**

Syntax

```
SQLRETURN   SQLGetEnvAttr    (
               SQLHENV          EnvironmentHandle,  /* henv */
               SQLINTEGER       Attribute,
               SQLPOINTER       ValuePtr,           /* Value */
               SQLINTEGER       BufferLength,
               SQLINTEGER       *StringLengthPtr);  /* StringLength */
```

Function arguments

Table 89. SQLGetEnvAttr arguments

Data type	Argument	Use	Description
SQLHENV	*EnvironmentHandle*	input	Environment handle.
SQLINTEGER	*Attribute*	input	Attribute to receive. Refer to the list of environment attributes and their descriptions.
SQLPOINTER	*ValuePtr*	output	A pointer to memory in which to return the current value of the attribute specified by *Attribute*.
SQLINTEGER	*BufferLength*	input	Maximum size of buffer pointed to by *ValuePtr*, if the attribute value is a character string; otherwise, ignored.
SQLINTEGER *	*StringLengthPtr*	output	Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes returned for the null-termination character) available to return in *ValuePtr*. If *ValuePtr* is a null pointer, no length is returned. If the attribute value is a character string, and the number of bytes available to return is greater than or equal to *BufferLength*, the data in *ValuePtr* is truncated to *BufferLength* minus the length of a null-termination character and is null-terminated by CLI.

If *Attribute* does not denote a string, then CLI ignores *BufferLength* and does not set *StringLengthPtr*.

Usage

SQLGetEnvAttr() can be called at any time between the allocation and freeing of the environment handle. It obtains the current value of the environment attribute.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 90. SQLGetEnvAttr SQLSTATEs

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY092	Option type out of range.	An invalid *Attribute* value was specified.

Restrictions

None.

Example

```
/* retrieve the current environment attribute value */
cliRC = SQLGetEnvAttr(henv, SQL_ATTR_OUTPUT_NTS, &output_nts, 0, NULL);
```

SQLGetFunctions function (CLI) - Get functions

Determines whether a specific CLI or ODBC function is supported.

This allows applications to adapt to varying levels of support when connecting to different database servers.

Specification:

- **CLI** 2.1
- **ODBC** 1.0
- **ISO CLI**

A connection to a database server must exist before calling this function.

Syntax

```
SQLRETURN   SQLGetFunctions  (
               SQLHDBC           ConnectionHandle,  /* hdbc */
               SQLUSMALLINT      FunctionId,        /* fFunction */
               SQLUSMALLINT      *SupportedPtr);    /* pfExists */
```

Function arguments

Table 91. SQLGetFunctions arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Database connection handle.
SQLUSMALLINT	*FunctionId*	input	The function being queried.
SQLUSMALLINT *	*SupportedPtr*	output	Pointer to location where this function will return SQL_TRUE or SQL_FALSE depending on whether the function being queried is supported.

Usage

If *FunctionId* is set to SQL_API_ALL_FUNCTIONS, then *SupportedPtr* must point to an SQLSMALLINT array of 100 elements. The array is indexed by the *FunctionId* values used to identify many of the functions. Some elements of the array are unused and reserved. Since some *FunctionId* values are greater than 100, the array method can not be used to obtain a list of functions. The SQLGetFunctions() call must be explicitly issued for all *FunctionId* values equal to or above 100. The complete set of *FunctionId* values is defined in `sqlcli1.h`.

Note: The LOB support functions (SQLGetLength(), SQLGetPosition(), SQLGetSubString(), SQLBindFileToCol(), SQLBindFileToCol()) are not supported when connected to IBM RDBMSs that do not support LOB data types.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 92. SQLGetFunctions SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	SQLGetFunctions() was called before a database connection was established.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Authorization

None.

Example

```
/* check to see if SQLGetInfo() is supported */
cliRC = SQLGetFunctions(hdbc, SQL_API_SQLGETINFO, &supported);
```

References

None.

SQLGetInfo function (CLI) - Get general information

Returns general information about the DBMS that the application is currently connected to.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

SQLGetInfo() returns general information about the database management system (DBMS) that the application is currently connected to.

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is `SQLGetInfoW()`. See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLGetInfo       (
               SQLHDBC           ConnectionHandle,   /* hdbc */
               SQLUSMALLINT      InfoType,           /* fInfoType */
               SQLPOINTER        InfoValuePtr,       /* rgbInfoValue */
               SQLSMALLINT       BufferLength,       /* cbInfoValueMax */
               SQLSMALLINT       *StringLengthPtr);  /* pcbInfoValue */
```

Function arguments

Table 93. SQLGetInfo arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	Input	The database connection handle.
SQLUSMALLINT	*InfoType*	Input	The type of information that is required. The possible values for this argument are described in Information returned by SQLGetInfo().
SQLPOINTER	*InfoValuePtr*	Output and input	Pointer to buffer where this function stores the information that you want. Depending on the type of information that is being retrieved, 5 types of information can be returned: • 16-bit integer value • 32-bit integer value • 32-bit binary value • 32-bit mask • Null-terminated character string If the *InfoType* argument is SQL_DRIVER_HDESC or SQL_DRIVER_HSTMT, *InfoValuePtr* is both input and output argument.

Table 93. SQLGetInfo arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*BufferLength*	Input	The maximum length of the buffer pointed by *InfoValuePtr* pointer. If **InfoValuePtr* is a Unicode string, the *BufferLength* argument must be an even number.
SQLSMALLINT *	*StringLengthPtr*	Output	Pointer to location where this function returns the total number of bytes of information that is available to return. For string output, the length does not include the null terminating character. If the value in the location pointed by *StringLengthPtr* is greater than the size specified in *BufferLength*, the string output information would be truncated to *BufferLength* - 1 bytes and the function returns with SQL_SUCCESS_WITH_INFO.

Usage

See Information returned by SQLGetInfo() for a list of the possible values of the *InfoType* argument and a description of the information that the `SQLGetInfo()` function would return for that value.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 94. SQLGetInfo SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The requested information is returned as a string, and its length exceeded the length of the application buffer as specified in the *BufferLength* argument. The *StringLengthPtr* argument contains the actual (not truncated) length of the requested information. (Function returns SQL_SUCCESS_WITH_INFO return code.)
08003	Connection is closed.	The type of information that is requested in the *InfoType* argument requires an open connection. Only the SQL_ODBC_VER information does not require an open connection.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY090	Invalid string or buffer length.	The value specified for the *BufferLength* argument is less than 0.
HY096	Information type out of range.	An invalid *InfoType* argument is specified.
HYC00	Driver not capable.	The value specified in the *InfoType* argument is not supported by either CLI or the data source.

Restrictions

None.

Example

```
/* get server name information */
cliRC = SQLGetInfo(hdbc, SQL_DBMS_NAME, imageInfoBuf, 255, &outlen);

/* ... */

/* get client driver name information */
cliRC = SQLGetInfo(hdbc, SQL_DRIVER_NAME, imageInfoBuf, 255, &outlen);
```

Information returned by SQLGetInfo()

Note: CLI returns a value for each *InfoType* argument in this table. If the *InfoType* argument does not apply or is not supported, the result is dependent on the return type. If the return type is a:
- Character string ("Y" or "N"), "N" is returned.
- Character string (not "Y" or "N"), an empty string is returned.
- 32-bit integer, 0 (zero) is returned.
- 32-bit mask, 0 (zero) is returned.

SQL_ACCESSIBLE_PROCEDURES (string)
A character string of "Y" indicates that you can run all procedures that are returned by the function `SQLProcedures()`. "N" indicates there might be procedures returned that you cannot run.

SQL_ACCESSIBLE_TABLES (string)
A character string of "Y" indicates that you are guaranteed SELECT privilege to all tables that are returned by the function `SQLTables()`. "N" indicates that there might be tables returned that you cannot access.

SQL_AGGREGATE_FUNCTIONS (32-bit mask)
A bit mask that enumerates support for the listed aggregation functions:
- SQL_AF_ALL
- SQL_AF_AVG
- SQL_AF_COUNT
- SQL_AF_DISTINCT
- SQL_AF_MAX
- SQL_AF_MIN
- SQL_AF_SUM

SQL_ALTER_DOMAIN (32-bit mask)
CLI returns 0 that indicates that the ALTER DOMAIN statement is not supported.

ODBC also defines the listed values that are not returned by CLI:
- SQL_AD_ADD_CONSTRAINT_DEFERRABLE
- SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE
- SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED
- SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE
- SQL_AD_ADD_DOMAIN_CONSTRAINT
- SQL_AD_ADD_DOMAIN_DEFAULT
- SQL_AD_CONSTRAINT_NAME_DEFINITION
- SQL_AD_DROP_DOMAIN_CONSTRAINT
- SQL_AD_DROP_DOMAIN_DEFAULT

SQL_ALTER_TABLE (32-bit mask)
Indicates which clauses in the ALTER TABLE statement are supported by the DBMS.
- SQL_AT_ADD_COLUMN_COLLATION
- SQL_AT_ADD_COLUMN_DEFAULT
- SQL_AT_ADD_COLUMN_SINGLE
- SQL_AT_ADD_CONSTRAINT
- SQL_AT_ADD_TABLE_CONSTRAINT
- SQL_AT_CONSTRAINT_NAME_DEFINITION
- SQL_AT_DROP_COLUMN_CASCADE
- SQL_AT_DROP_COLUMN_DEFAULT
- SQL_AT_DROP_COLUMN_RESTRICT
- SQL_AT_DROP_TABLE_CONSTRAINT_CASCADE
- SQL_AT_DROP_TABLE_CONSTRAINT_RESTRICT
- SQL_AT_SET_COLUMN_DEFAULT
- SQL_AT_CONSTRAINT_INITIALLY_DEFERRED
- SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE
- SQL_AT_CONSTRAINT_DEFERRABLE
- SQL_AT_CONSTRAINT_NON_DEFERRABLE

SQL_APPLICATION_CODEPAGE (32-bit unsigned integer)
Indicates the application code page.

SQL_ASYNC_MODE (32-bit unsigned integer)
Indicates the level of asynchronous support in the driver:
- SQL_AM_CONNECTION : Connection level asynchronous execution is supported. Either all statement handles that are associated with a given connection handle are in asynchronous mode, or all are in synchronous mode. A statement handle that is on a connection cannot be in asynchronous mode while another statement handle on the same connection is in synchronous mode, and vice versa.
- SQL_AM_STATEMENT : Statement level asynchronous execution is supported. Some statement handles that are associated with a connection handle can be in asynchronous mode, while other statement handles on the same connection are in synchronous mode.
- SQL_AM_NONE : Asynchronous mode is not supported.

  This value is also returned if the CLI/ODBC configuration keyword ASYNCENABLE is set to disable asynchronous execution.

SQL_BATCH_ROW_COUNT (32-bit mask)
Indicates how row counts are dealt with. CLI always returns SQL_BRC_ROLLED_UP, which indicates that row counts for consecutive INSERT, DELETE, or UPDATE statements are rolled into one.

ODBC also defines the values that are not returned by CLI:
- SQL_BRC_PROCEDURES
- SQL_BRC_EXPLICIT

SQL_BATCH_SUPPORT (32-bit mask)
Indicates which levels of batches are supported:
- SQL_BS_SELECT_EXPLICIT : Supports explicit batches that can have result-set generating statements.
- SQL_BS_ROW_COUNT_EXPLICIT : Supports explicit batches that can have row-count generating statements.
- SQL_BS_SELECT_PROC : Supports explicit procedures that can have result-set generating statements.
- SQL_BS_ROW_COUNT_PROC : Supports explicit procedures that can have row-count generating statements.

SQL_BOOKMARK_PERSISTENCE (32-bit mask)
Indicates when bookmarks remain valid after an operation:
- SQL_BP_CLOSE : After an application calls `SQLFreeStmt()` with the SQL_CLOSE option, or `SQLCloseCursor()` to close the cursor associated with a statement.
- SQL_BP_DELETE : After that row has been deleted.
- SQL_BP_DROP : Bookmarks are valid after an application calls `SQLFreeHandle()` with a *HandleType* of SQL_HANDLE_STMT to drop a statement.
- SQL_BP_TRANSACTION : After an application commits or rolls back a transaction.
- SQL_BP_UPDATE : After any column in that row has been updated, including key columns.
- SQL_BP_OTHER_HSTMT : A bookmark that is associated with one statement can be used with another statement. Unless SQL_BP_CLOSE or SQL_BP_DROP is specified, the cursor on the first statement must be open.

SQL_CATALOG_LOCATION (16-bit integer)
A 16-bit integer value that indicates the position of the qualifier in a qualified table name. CLI always returns SQL_CL_START for this information type. ODBC also defines the value SQL_CL_END which is not returned by CLI.

In previous versions of CLI this *InfoType* was SQL_QUALIFIER_LOCATION.

SQL_CATALOG_NAME (string)
A character string of "Y" indicates that the server supports catalog names. "N" indicates that catalog names are not supported.

SQL_CATALOG_NAME_SEPARATOR (string)
The character(s) used as a separator between a catalog name and the qualified name element that follows or precedes it.

In previous versions of CLI this *InfoType* was SQL_QUALIFIER_NAME_SEPARATOR.

SQL_CATALOG_TERM (string)
The terminology of the database vendor for a qualifier (catalog).

The name that the vendor uses for the high-order part of a three part name.

If the target DBMS does not support three-part naming, a zero-length string is returned.

In previous versions of CLI this *InfoType* was SQL_QUALIFIER_TERM.

SQL_CATALOG_USAGE (32-bit mask)

A 32-bit mask enumerating the statements in which you can use catalogs. The SQL_CATALOG_USAGE is similar to SQL_SCHEMA_USAGE, except that SQL_CATALOG_USAGE is specific for catalogs.
- SQL_CU_DML_STATEMENTS : All data manipulation language (DML) statements.
- SQL_CU_INDEX_DEFINITION : All index definition statements.
- SQL_CU_PRIVILEGE_DEFINITION : All privilege definition statements.
- SQL_CU_PROCEDURE_INVOCATION : The ODBC procedure invocation statement.

- SQL_CU_TABLE_DEFINITION : All table definition statements.

A value of 0 is returned if catalogs are not supported by the data source.

In previous versions of CLI, this *InfoType* argument was SQL_QUALIFIER_USAGE.

SQL_COLLATION_SEQ (string)
Indicates the name of the default collation sequence for the default character set for this server (for example ISO 8859-1 or EBCDIC). If the collation sequence is unknown, an empty string is returned.

SQL_COLUMN_ALIAS (string)
Returns "Y" if column aliases are supported, or "N" if they are not.

SQL_CONCAT_NULL_BEHAVIOR (16-bit integer)
Indicates how the concatenation of NULL valued character data type columns with non-NULL valued character data type columns is handled.
- SQL_CB_NULL : A NULL value (this behavior is the case for IBM RDBMS).
- SQL_CB_NON_NULL : A concatenation of non-NULL column values.

SQL_CONVERT_* (32-bit masks)
SQL_CONVERT_BIGINT (32-bit mask)
SQL_CONVERT_BINARY (32-bit mask)
SQL_CONVERT_BIT (32-bit mask)
SQL_CONVERT_CHAR (32-bit mask)
SQL_CONVERT_DATE (32-bit mask)
SQL_CONVERT_DECIMAL (32-bit mask)
SQL_CONVERT_DOUBLE (32-bit mask)
SQL_CONVERT_FLOAT (32-bit mask)
SQL_CONVERT_INTEGER (32-bit mask)
SQL_CONVERT_INTERVAL_YEAR_MONTH (32-bit mask)
SQL_CONVERT_INTERVAL_DAY_TIME (32-bit mask)
SQL_CONVERT_LONGVARBINARY (32-bit mask)
SQL_CONVERT_LONGVARCHAR (32-bit mask)
SQL_CONVERT_NUMERIC (32-bit mask)
SQL_CONVERT_REAL (32-bit mask)
SQL_CONVERT_SMALLINT (32-bit mask)
SQL_CONVERT_TIME (32-bit mask)
SQL_CONVERT_TIMESTAMP (32-bit mask)
SQL_CONVERT_TINYINT (32-bit mask)
SQL_CONVERT_VARBINARY (32-bit mask)
SQL_CONVERT_VARCHAR (32-bit mask)
SQL_CONVERT_WCHAR (32-bit mask)
SQL_CONVERT_WLONGVARCHAR (32-bit mask)
SQL_CONVERT_WVARCHAR (32-bit mask)

Indicates the conversions that are supported by the data source with the CONVERT scalar function for data of the type named in the *InfoType*. If the bit mask equals zero, the data source does not support any conversions for the named data type, including conversions to the same data type.
For example, to find out if a data source supports the conversion of SQL_INTEGER data to the SQL_DECIMAL data type, an application calls `SQLGetInfo()` function with the *InfoType* argument of SQL_CONVERT_INTEGER. The application then performs `AND` operation on the returned bit mask with SQL_CVT_DECIMAL. If the resulting value

is nonzero, the conversion is supported.
The listed bit masks are used to determine which conversions are supported:
- SQL_CVT_BIGINT
- SQL_CVT_BINARY
- SQL_CVT_BIT
- SQL_CVT_CHAR
- SQL_CVT_DATE
- SQL_CVT_DECIMAL
- SQL_CVT_DOUBLE
- SQL_CVT_FLOAT
- SQL_CVT_INTEGER
- SQL_CVT_INTERVAL_YEAR_MONTH
- SQL_CVT_INTERVAL_DAY_TIME
- SQL_CVT_LONGVARBINARY
- SQL_CVT_LONGVARCHAR
- SQL_CVT_NUMERIC
- SQL_CVT_REAL
- SQL_CVT_SMALLINT
- SQL_CVT_TIME
- SQL_CVT_TIMESTAMP
- SQL_CVT_TINYINT
- SQL_CVT_VARBINARY
- SQL_CVT_VARCHAR
- SQL_CVT_WCHAR
- SQL_CVT_WLONGVARCHAR
- SQL_CVT_WVARCHAR

SQL_CONNECT_CODEPAGE (32-bit unsigned integer)
Indicates the code page of the current connection.

SQL_CONVERT_FUNCTIONS (32-bit mask)
Indicates the scalar conversion functions that are supported by the driver and associated data source.

CLI Version 2.1.1 and later supports ODBC scalar conversions between char variables (CHAR, VARCHAR, LONG VARCHAR, and CLOB) and DOUBLE (or FLOAT).
- SQL_FN_CVT_CONVERT : Used to determine which conversion functions are supported.

SQL_CORRELATION_NAME (16-bit integer)
Indicates the degree of correlation name support by the server:
- SQL_CN_ANY : Any valid user-defined name is supported.
- SQL_CN_NONE : Correlation name is not supported.
- SQL_CN_DIFFERENT : Correlation name is supported, but it must be different than the name of the table that it represents.

SQL_CREATE_ASSERTION (32-bit mask)
Indicates which clauses in the CREATE ASSERTION statement are supported by the DBMS. CLI always returns zero; the CREATE ASSERTION statement is not supported.

ODBC also defines the listed values that are not returned by CLI:
- SQL_CA_CREATE_ASSERTION
- SQL_CA_CONSTRAINT_INITIALLY_DEFERRED
- SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE
- SQL_CA_CONSTRAINT_DEFERRABLE
- SQL_CA_CONSTRAINT_NON_DEFERRABLE

SQL_CREATE_CHARACTER_SET (32-bit mask)
Indicates which clauses in the CREATE CHARACTER SET statement are supported by the DBMS. CLI always returns zero; the CREATE CHARACTER SET statement is not supported.

ODBC also defines the listed values that are not returned by CLI:
- SQL_CCS_CREATE_CHARACTER_SET
- SQL_CCS_COLLATE_CLAUSE
- SQL_CCS_LIMITED_COLLATION

SQL_CREATE_COLLATION (32-bit mask)
Indicates which clauses in the CREATE COLLATION statement are supported by the DBMS. CLI always returns zero; the CREATE COLLATION statement is not supported.

ODBC also defines the listed values that are not returned by CLI:
- SQL_CCOL_CREATE_COLLATION

SQL_CREATE_DOMAIN (32-bit mask)
Indicates which clauses in the CREATE DOMAIN statement are supported by the DBMS. CLI always returns zero; the CREATE DOMAIN statement is not supported.

ODBC also defines the listed values that are not returned by CLI:
- SQL_CDO_CREATE_DOMAIN
- SQL_CDO_CONSTRAINT_NAME_DEFINITION
- SQL_CDO_DEFAULT
- SQL_CDO_CONSTRAINT
- SQL_CDO_COLLATION
- SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED
- SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE
- SQL_CDO_CONSTRAINT_DEFERRABLE
- SQL_CDO_CONSTRAINT_NON_DEFERRABLE

SQL_CREATE_MODULE (32-bit mask)
Indicates which clauses in the CREATE MODULE statement are supported by the DBMS.CLI always returns zero for DB2 for z/OS.

CLI returns the listed values:
- SQL_CM_CREATE_MODULE
- SQL_CM_AUTHORIZATION
- SQL_CM_DEFAULT_CHARACTER_SET

SQL_CREATE_SCHEMA (32-bit mask)
Indicates which clauses in the CREATE SCHEMA statement are supported by the DBMS:
- SQL_CS_CREATE_SCHEMA
- SQL_CS_AUTHORIZATION
- SQL_CS_DEFAULT_CHARACTER_SET

SQL_CREATE_TABLE (32-bit mask)
Indicates which clauses in the CREATE TABLE statement are supported by the DBMS.

The lilsted bit masks are used to determine which clauses are supported:
- SQL_CT_CREATE_TABLE
- SQL_CT_TABLE_CONSTRAINT
- SQL_CT_CONSTRAINT_NAME_DEFINITION

The listed bits specify the ability to create temporary tables:
- SQL_CT_COMMIT_PRESERVE : Deleted rows are preserved on commit.

- SQL_CT_COMMIT_DELETE : Deleted rows are deleted on commit.
- SQL_CT_GLOBAL_TEMPORARY : Global temporary tables can be created.
- SQL_CT_LOCAL_TEMPORARY : Local temporary tables can be created.

The listed bits specify the ability to create column constraints:
- SQL_CT_COLUMN_CONSTRAINT : Specifying column constraints is supported.
- SQL_CT_COLUMN_DEFAULT : Specifying column defaults is supported.
- SQL_CT_COLUMN_COLLATION : Specifying column collation is supported.

The listed bits specify the supported constraint attributes, if specifying column or table constraints is supported:
- SQL_CT_CONSTRAINT_INITIALLY_DEFERRED
- SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE
- SQL_CT_CONSTRAINT_DEFERRABLE
- SQL_CT_CONSTRAINT_NON_DEFERRABLE

SQL_CREATE_TRANSLATION (32-bit mask)
Indicates which clauses in the CREATE TRANSLATION statement are supported by the DBMS. CLI always returns zero; the CREATE TRANSLATION statement is not supported.

ODBC also defines the listed value that is not returned by CLI:
- SQL_CTR_CREATE_TRANSLATION

SQL_CREATE_VIEW (32-bit mask)
Indicates which clauses in the CREATE VIEW statement are supported by the DBMS:
- SQL_CV_CREATE_VIEW
- SQL_CV_CHECK_OPTION
- SQL_CV_CASCADED
- SQL_CV_LOCAL

A return value of 0 means that the CREATE VIEW statement is not supported.

SQL_CURSOR_COMMIT_BEHAVIOR (16-bit integer)
Indicates how a COMMIT operation affects cursors. A value of:
- SQL_CB_DELETE, deletes cursors and drops access plans for dynamic SQL statements.
- SQL_CB_CLOSE, deletes cursors, but retains access plans for dynamic SQL statements (including non-query statements)
- SQL_CB_PRESERVE, retains cursors and access plans for dynamic statements (including non-query statements). Applications can continue to fetch data, or close the cursor and re-execute the query without preparing again the statement.

  Note: After COMMIT, a FETCH must be issued to reposition the cursor before actions such as positioned updates or deletes can be taken.

SQL_CURSOR_ROLLBACK_BEHAVIOR (16-bit integer)
Indicates how a ROLLBACK operation affects cursors. A value of:
- SQL_CB_DELETE, deletes cursors and drops access plans for dynamic SQL statements.
- SQL_CB_CLOSE, deletes cursors, but retains access plans for dynamic SQL statements (including non-query statements)

- SQL_CB_PRESERVE, retains cursors and access plans for dynamic statements (including non-query statements). Applications can continue to fetch data, or close the cursor and re-execute the query without preparing again the statement.

  Note: DB2 servers do not have the SQL_CB_PRESERVE property.

SQL_CURSOR_SENSITIVITY (32-bit unsigned integer)
Indicates support for cursor sensitivity:

- SQL_INSENSITIVE, all cursors on the statement handle show the result set without reflecting any changes made to it by any other cursor within the same transaction.
- SQL_UNSPECIFIED, it is unspecified whether cursors on the statement handle make visible the changes made to a result set by another cursor within the same transaction. Cursors on the statement handle might make visible none, some, or all such changes.
- SQL_SENSITIVE, cursors are sensitive to changes made by other cursors within the same transaction.

SQL_DATA_SOURCE_NAME (string)
Indicates the data source name used during connection. If the application called `SQLConnect()`, this character string is the value of the *szDSN* argument. If the application called `SQLDriverConnect()` or `SQLBrowseConnect()`, this character string is the value of the DSN keyword in the connection string passed to the driver. If the connection string did not contain the DSN keyword, this character string is an empty string.

SQL_DATA_SOURCE_READ_ONLY (string)
A character string of "Y" indicates that the database is set to READ ONLY mode, "N" indicates that is not set to READ ONLY mode. This characteristic pertains only to the data source itself; it is not characteristic of the driver that enables access to the data source.

SQL_DATABASE_CODEPAGE (32-bit unsigned integer)
Indicates the code page of the database that the application is currently connected to.

SQL_DATABASE_NAME (string)
The name of the current database in use

Note: This string is the same as that returned by the SELECT CURRENT SERVER statement on non-host systems. For host databases, such as DB2 for z/OS or DB2 for i, the string returned is the DCS database name. This database name was provided when the CATALOG DCS DATABASE DIRECTORY command was issued at the DB2 Connect gateway.

SQL_DATETIME_LITERALS (32-bit unsigned integer)
Indicates the datetime literals that are supported by the DBMS. CLI always returns zero; datetime literals are not supported.

ODBC also defines the listed values that are not returned by CLI:

- SQL_DL_SQL92_DATE
- SQL_DL_SQL92_TIME
- SQL_DL_SQL92_TIMESTAMP
- SQL_DL_SQL92_INTERVAL_YEAR
- SQL_DL_SQL92_INTERVAL_MONTH
- SQL_DL_SQL92_INTERVAL_DAY
- SQL_DL_SQL92_INTERVAL_HOUR
- SQL_DL_SQL92_INTERVAL_MINUTE

- SQL_DL_SQL92_INTERVAL_SECOND
- SQL_DL_SQL92_INTERVAL_YEAR_TO_MONTH
- SQL_DL_SQL92_INTERVAL_DAY_TO_HOUR
- SQL_DL_SQL92_INTERVAL_DAY_TO_MINUTE
- SQL_DL_SQL92_INTERVAL_DAY_TO_SECOND
- SQL_DL_SQL92_INTERVAL_HOUR_TO_MINUTE
- SQL_DL_SQL92_INTERVAL_HOUR_TO_SECOND
- SQL_DL_SQL92_INTERVAL_MINUTE_TO_SECOND

SQL_DBMS_NAME (string)
The name of the DBMS product being accessed

For example:
- "DB2/6000"
- "DB2/2"

SQL_DBMS_VER (string)
The Version of the DBMS product accessed. A string of the form 'mm.vv.rrrr' where *mm* is the major version, *vv* is the minor version, and *rrrr* is the release number. For example, "0r.01.0000" translates to major version r, minor version 1, release 0.

SQL_DDL_INDEX (32-bit unsigned integer)
Indicates support for the creation and dropping of indexes:
- SQL_DI_CREATE_INDEX
- SQL_DI_DROP_INDEX

SQL_DEFAULT_TXN_ISOLATION (32-bit mask)
The default transaction isolation level supported

One of the lilsted masks are returned:

- SQL_TXN_READ_UNCOMMITTED : Changes are immediately perceived by all transactions (dirty read, non-repeatable read, and phantoms are possible).

  This behavior is equivalent to Uncommitted Read level for IBM databases.
- SQL_TXN_READ_COMMITTED : Row read by transaction 1 can be altered and committed by transaction 2 (non-repeatable read and phantoms are possible)

  This behavior is equivalent to Cursor Stability level in IBM databases.
- SQL_TXN_REPEATABLE_READ : A transaction can add or remove rows matching the search condition or a pending transaction (repeatable read, but phantoms are possible)

  This behavior is equivalent to Read Stability level in IBM databases.
- SQL_TXN_SERIALIZABLE : Data affected by pending transaction is not available to other transactions (repeatable read, phantoms are not possible)

  This behavior is equivalent to Repeatable Read level in IBM databases.
- SQL_TXN_VERSIONING : Not applicable to IBM DBMSs.
- SQL_TXN_NOCOMMIT : Any changes are effectively committed at the end of a successful operation; no explicit commit or rollback is allowed.

  This is a IBM DB2 for IBM i isolation level.

In IBM terminology,
- SQL_TXN_READ_UNCOMMITTED is Uncommitted Read;
- SQL_TXN_READ_COMMITTED is Cursor Stability;
- SQL_TXN_REPEATABLE_READ is Read Stability;

- SQL_TXN_SERIALIZABLE is Repeatable Read.

SQL_DESCRIBE_PARAMETER (string)
"Y" if parameters can be described; "N" if not.

SQL_DM_VER (string)
Reserved.

SQL_DRIVER_BLDLEVEL
Build level information about the current version of CLI.

The information is in the listed format: s*YYMMDD*, where *YY* is the year of the build, *MM* is the month and *DD* is the day. For example, **s100610**.

For special builds, the format is: special_*JOBID*, where *JOBID* is the special build's job identification. For example, **special_39899**.

For full version information, use SQL_DRIVER_BLDLEVEL with SQL_DRIVER_VER.

SQL_DRIVER_HDBC (32 bits)
CLI's database handle

SQL_DRIVER_HDESC (32 bits)
CLI's descriptor handle

SQL_DRIVER_HENV (32 bits)
CLI's environment handle

SQL_DRIVER_HLIB (32 bits)
Reserved.

SQL_DRIVER_HSTMT (32 bits)
CLI's statement handle

In an ODBC environment with an ODBC Driver Manager, if *InfoType* is set to SQL_DRIVER_HSTMT, the Driver Manager statement handle (the one returned from `SQLAllocStmt()`) must be passed on input in *rgbInfoValue* from the application. In this case *rgbInfoValue* is both an input and an output argument. The ODBC Driver Manager is responsible for returning the mapped value. ODBC applications wishing to call CLI specific functions (such as the LOB functions) can access them, by passing these handle values to the functions after loading the CLI library and issuing an operating system function call to invoke the required functions.

SQL_DRIVER_NAME (string)
The file name of the CLI implementation.

SQL_DRIVER_ODBC_VER (string)
The version number of ODBC that CLI supports. By Default CLI returns “03.51”. You can call the `SQLSetEnvAttr()` function to change the ODBC driver version. If you set the SQL_ATTR_ODBC_VERSION attribute to SQL_OV_ODBC3_80 (value 380), CLI returns “03.80”.

SQL_DRIVER_VER (string)
The version of the IBM Data Server Driver for ODBC and CLI. A string of the form 'mm.vv.rrrr' where *mm* is the major version, *vv* is the minor version, and *rrrr* is the release. For example, "05.01.0000" translates to major version 5, minor version 1, release 0. For full version information, use SQL_DRIVER_VER with SQL_DRIVER_BLDLEVEL.

SQL_DROP_ASSERTION (32-bit unsigned integer)
Indicates which clause in the DROP ASSERTION statement is supported by the DBMS. CLI always returns zero; the DROP ASSERTION statement is not supported.

ODBC also defines the SQL_DA_DROP_ASSERTION value that is not returned by CLI.

SQL_DROP_CHARACTER_SET (32-bit unsigned integer)
Indicates which clause in the DROP CHARACTER SET statement is supported by the DBMS. CLI always returns zero; the DROP CHARACTER SET statement is not supported.

ODBC also defines the SQL_DCS_DROP_CHARACTER_SET value that is not returned by CLI.

SQL_DROP_COLLATION (32-bit unsigned integer)
Indicates which clause in the DROP COLLATION statement is supported by the DBMS. CLI always returns zero; the DROP COLLATION statement is not supported.

ODBC also defines the SQL_DC_DROP_COLLATION value that is not returned by CLI.

SQL_DROP_DOMAIN (32-bit unsigned integer)
Indicates which clauses in the DROP DOMAIN statement are supported by the DBMS. CLI always returns zero; the DROP DOMAIN statement is not supported.

ODBC also defines the listed values that are not returned by CLI:
- SQL_DD_DROP_DOMAIN
- SQL_DD_CASCADE
- SQL_DD_RESTRICT

SQL_DROP_MODULE (32-bit unsigned integer)
Indicates which clauses in the DROP MODULE statement are supported by the DBMS. CLI always returns zero for DB2 for z/OS.

CLI returns the listed values:
- SQL_DM_DROP_MODULE
- SQL_DM_RESTRICT

SQL_DROP_SCHEMA (32-bit unsigned integer)
Indicates which clauses in the DROP SCHEMA statement are supported by the DBMS. CLI always returns zero; the DROP SCHEMA statement is not supported.

ODBC also defines the listed values that are not returned by CLI:
- SQL_DS_CASCADE
- SQL_DS_RESTRICT

SQL_DROP_TABLE (32-bit unsigned integer)
Indicates which clauses in the DROP TABLE statement are supported by the DBMS. Valid returned values are:
- SQL_DT_DROP_TABLE
- SQL_DT_CASCADE
- SQL_DT_RESTRICT

SQL_DROP_TRANSLATION (32-bit unsigned integer)
Indicates which clauses in the DROP TRANSLATION statement are supported by the DBMS. CLI always returns zero; the DROP TRANSLATION statement is not supported.

ODBC also defines the listed value that is not returned by CLI:
- SQL_DTR_DROP_TRANSLATION

SQL_DROP_VIEW (32-bit unsigned integer)
Indicates which clauses in the DROP VIEW statement are supported by the DBMS. CLI always returns zero; the DROP VIEW statement is not supported.

ODBC also defines the listed values that are not returned by CLI:
- SQL_DV_CASCADE
- SQL_DV_RESTRICT

SQL_DTC_TRANSITION_COST (32-bit unsigned mask)
Used by Microsoft Transaction Server to determine whether the enlistment process for a connection is expensive. CLI returns:
- SQL_DTC_ENLIST_EXPENSIVE
- SQL_DTC_UNENLIST_EXPENSIVE

SQL_DYNAMIC_CURSOR_ATTRIBUTES1 (32-bit mask)
Indicates the attributes of a dynamic cursor that are supported by CLI (subset 1 of 2). Valid returned values are:
- SQL_CA1_NEXT
- SQL_CA1_ABSOLUTE
- SQL_CA1_RELATIVE
- SQL_CA1_BOOKMARK
- SQL_CA1_LOCK_EXCLUSIVE
- SQL_CA1_LOCK_NO_CHANGE
- SQL_CA1_LOCK_UNLOCK
- SQL_CA1_POS_POSITION
- SQL_CA1_POS_UPDATE
- SQL_CA1_POS_DELETE
- SQL_CA1_POS_REFRESH
- SQL_CA1_POSITIONED_UPDATE
- SQL_CA1_POSITIONED_DELETE
- SQL_CA1_SELECT_FOR_UPDATE
- SQL_CA1_BULK_ADD
- SQL_CA1_BULK_UPDATE_BY_BOOKMARK
- SQL_CA1_BULK_DELETE_BY_BOOKMARK
- SQL_CA1_BULK_FETCH_BY_BOOKMARK

SQL_DYNAMIC_CURSOR_ATTRIBUTES2 (32-bit mask)
Indicates the attributes of a dynamic cursor that are supported by CLI (subset 2 of 2). Valid returned values are:
- SQL_CA2_READ_ONLY_CONCURRENCY
- SQL_CA2_LOCK_CONCURRENCY
- SQL_CA2_OPT_ROWVER_CONCURRENCY
- SQL_CA2_OPT_VALUES_CONCURRENCY
- SQL_CA2_SENSITIVITY_ADDITIONS
- SQL_CA2_SENSITIVITY_DELETIONS
- SQL_CA2_SENSITIVITY_UPDATES
- SQL_CA2_MAX_ROWS_SELECT
- SQL_CA2_MAX_ROWS_INSERT
- SQL_CA2_MAX_ROWS_DELETE
- SQL_CA2_MAX_ROWS_UPDATE
- SQL_CA2_MAX_ROWS_CATALOG
- SQL_CA2_MAX_ROWS_AFFECTS_ALL
- SQL_CA2_CRC_EXACT
- SQL_CA2_CRC_APPROXIMATE

- SQL_CA2_SIMULATE_NON_UNIQUE
- SQL_CA2_SIMULATE_TRY_UNIQUE
- SQL_CA2_SIMULATE_UNIQUE

SQL_EXPRESSIONS_IN_ORDERBY (string)
The character string "Y" indicates that the database server supports the DIRECT specification of expressions in the ORDER BY list, "N" indicates that it does not.

SQL_FETCH_DIRECTION (32-bit mask)
The supported fetch directions.

The listed bit masks are used with the flag to determine which options are supported:
- SQL_FD_FETCH_NEXT
- SQL_FD_FETCH_FIRST
- SQL_FD_FETCH_LAST
- SQL_FD_FETCH_PREV
- SQL_FD_FETCH_ABSOLUTE
- SQL_FD_FETCH_RELATIVE
- SQL_FD_FETCH_RESUME

SQL_FILE_USAGE (16-bit integer)
Indicates how a single-tier driver directly treats files in a data source. The IBM Data Server Driver for ODBC and CLI driver is not a single-tier driver, and therefore always returns SQL_FILE_NOT_SUPPORTED.

ODBC also defines the listed values that are not returned by CLI:
- SQL_FILE_TABLE
- SQL_FILE_CATALOG

SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1 (32-bit mask)
Indicates the attributes of a forward-only cursor that are supported by CLI. Valid returned values are (subset 1 of 2):
- SQL_CA1_NEXT
- SQL_CA1_POSITIONED_UPDATE
- SQL_CA1_POSITIONED_DELETE
- SQL_CA1_SELECT_FOR_UPDATE
- SQL_CA1_LOCK_EXCLUSIVE
- SQL_CA1_LOCK_NO_CHANGE
- SQL_CA1_LOCK_UNLOCK
- SQL_CA1_POS_POSITION
- SQL_CA1_POS_UPDATE
- SQL_CA1_POS_DELETE
- SQL_CA1_POS_REFRESH
- SQL_CA1_BULK_ADD
- SQL_CA1_BULK_UPDATE_BY_BOOKMARK
- SQL_CA1_BULK_DELETE_BY_BOOKMARK
- SQL_CA1_BULK_FETCH_BY_BOOKMARK

SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2 (32-bit mask)
Indicates the attributes of a forward-only cursor that are supported by CLI. Valid returned values are (subset 2 of 2):
- SQL_CA2_READ_ONLY_CONCURRENCY
- SQL_CA2_LOCK_CONCURRENCY
- SQL_CA2_MAX_ROWS_SELECT
- SQL_CA2_MAX_ROWS_CATALOG
- SQL_CA2_OPT_ROWVER_CONCURRENCY
- SQL_CA2_OPT_VALUES_CONCURRENCY

- SQL_CA2_SENSITIVITY_ADDITIONS
- SQL_CA2_SENSITIVITY_DELETIONS
- SQL_CA2_SENSITIVITY_UPDATES
- SQL_CA2_MAX_ROWS_INSERT
- SQL_CA2_MAX_ROWS_DELETE
- SQL_CA2_MAX_ROWS_UPDATE
- SQL_CA2_MAX_ROWS_AFFECTS_ALL
- SQL_CA2_CRC_EXACT
- SQL_CA2_CRC_APPROXIMATE
- SQL_CA2_SIMULATE_NON_UNIQUE
- SQL_CA2_SIMULATE_TRY_UNIQUE
- SQL_CA2_SIMULATE_UNIQUE

SQL_GETDATA_EXTENSIONS (32-bit mask)
Indicates whether extensions to the `SQLGetData()` function are supported. The listed extensions are currently identified and supported by CLI:
- SQL_GD_ANY_COLUMN, `SQLGetData()` can be called for unbound columns that precede the last bound column.
- SQL_GD_ANY_ORDER, `SQLGetData()` can be called for columns in any order.

ODBC also defines the listed extensions which are not returned by CLI:
- SQL_GD_BLOCK
- SQL_GD_BOUND

SQL_GROUP_BY (16-bit integer)
Indicates the degree of support for the GROUP BY clause by the server. Valid returned values are:
- SQL_GB_NO_RELATION - No relationship between the columns in the GROUP BY clause and in the SELECT list.
- SQL_GB_NOT_SUPPORTED - GROUP BY clause not supported.
- SQL_GB_GROUP_BY_EQUALS_SELECT - GROUP BY clause must include all non-aggregated columns in the SELECT list.
- SQL_GB_GROUP_BY_CONTAINS_SELECT - GROUP BY clause must contain all non-aggregated columns in the SELECT list.
- SQL_GB_COLLATE - COLLATE clause can be specified at the end of each grouping column.

SQL_IDENTIFIER_CASE (16-bit integer)
Indicates the case sensitivity of object names (such as table-name).

Valid returned values are::
- SQL_IC_UPPER : Stored in uppercase.
- SQL_IC_LOWER : Stored in lowercase.
- SQL_IC_SENSITIVE : Case sensitive, stored in mixed-case.
- SQL_IC_MIXED : Not case sensitive, stored in mixed-case.

Note: Identifier names in IBM DBMSs are not case sensitive.

SQL_IDENTIFIER_QUOTE_CHAR (string)
Indicates the character that is used to surround a delimited identifier.

SQL_INDEX_KEYWORDS (32-bit mask)
Indicates the supported keywords for the CREATE INDEX statement. Valid returned values are:
- SQL_IK_NONE - None of the keywords are supported.
- SQL_IK_ASC - ASC keyword is supported.
- SQL_IK_DESC - DESC keyword is supported.
- SQL_IK_ALL - All keywords are supported.

To see if the CREATE INDEX statement is supported, an application can call the `SQLGetInfo()` function with the SQL_DLL_INDEX *InfoType* argument.

SQL_INFO_SCHEMA_VIEWS (32-bit mask)
Indicates the views in the INFORMATION_SCHEMA that are supported. CLI always returns zero; no views in the INFORMATION_SCHEMA are supported.

ODBC also defines the listed values that are not returned by CLI:
- SQL_ISV_ASSERTIONS
- SQL_ISV_CHARACTER_SETS
- SQL_ISV_CHECK_CONSTRAINTS
- SQL_ISV_COLLATIONS
- SQL_ISV_COLUMN_DOMAIN_USAGE
- SQL_ISV_COLUMN_PRIVILEGES
- SQL_ISV_COLUMNS
- SQL_ISV_CONSTRAINT_COLUMN_USAGE
- SQL_ISV_CONSTRAINT_TABLE_USAGE
- SQL_ISV_DOMAIN_CONSTRAINTS
- SQL_ISV_DOMAINS
- SQL_ISV_KEY_COLUMN_USAGE
- SQL_ISV_REFERENTIAL_CONSTRAINTS
- SQL_ISV_SCHEMATA
- SQL_ISV_SQL_LANGUAGES
- SQL_ISV_TABLE_CONSTRAINTS
- SQL_ISV_TABLE_PRIVILEGES
- SQL_ISV_TABLES
- SQL_ISV_TRANSLATIONS
- SQL_ISV_USAGE_PRIVILEGES
- SQL_ISV_VIEW_COLUMN_USAGE
- SQL_ISV_VIEW_TABLE_USAGE
- SQL_ISV_VIEWS

SQL_INSERT_STATEMENT (32-bit mask)
Indicates support for INSERT statements. Valid returned values are:
- SQL_IS_INSERT_LITERALS
- SQL_IS_INSERT_SEARCHED
- SQL_IS_SELECT_INTO

SQL_INTEGRITY (string)
The "Y" character string indicates that the data source supports Integrity Enhanced Facility (IEF) in SQL89 and in X/Open XPG4 Embedded SQL, an "N" indicates it does not.

In previous versions of CLI this *InfoType* argument was SQL_ODBC_SQL_OPT_IEF.

SQL_KEYSET_CURSOR_ATTRIBUTES1 (32-bit mask)
Indicates the attributes of a keyset-driven cursor that are supported by CLI. Valid returned values are (subset 1 of 2):
- SQL_CA1_NEXT
- SQL_CA1_ABSOLUTE
- SQL_CA1_RELATIVE
- SQL_CA1_BOOKMARK
- SQL_CA1_LOCK_EXCLUSIVE
- SQL_CA1_LOCK_NO_CHANGE
- SQL_CA1_LOCK_UNLOCK
- SQL_CA1_POS_POSITION

- SQL_CA1_POS_UPDATE
- SQL_CA1_POS_DELETE
- SQL_CA1_POS_REFRESH
- SQL_CA1_POSITIONED_UPDATE
- SQL_CA1_POSITIONED_DELETE
- SQL_CA1_SELECT_FOR_UPDATE
- SQL_CA1_BULK_ADD
- SQL_CA1_BULK_UPDATE_BY_BOOKMARK
- SQL_CA1_BULK_DELETE_BY_BOOKMARK
- SQL_CA1_BULK_FETCH_BY_BOOKMARK

SQL_KEYSET_CURSOR_ATTRIBUTES2 (32-bit mask)
Indicates the attributes of a keyset-driven cursor that are supported by CLI. Valid returned values are (subset 2 of 2):

- SQL_CA2_READ_ONLY_CONCURRENCY
- SQL_CA2_LOCK_CONCURRENCY
- SQL_CA2_OPT_ROWVER_CONCURRENCY
- SQL_CA2_OPT_VALUES_CONCURRENCY
- SQL_CA2_SENSITIVITY_ADDITIONS
- SQL_CA2_SENSITIVITY_DELETIONS
- SQL_CA2_SENSITIVITY_UPDATES
- SQL_CA2_MAX_ROWS_SELECT
- SQL_CA2_MAX_ROWS_INSERT
- SQL_CA2_MAX_ROWS_DELETE
- SQL_CA2_MAX_ROWS_UPDATE
- SQL_CA2_MAX_ROWS_CATALOG
- SQL_CA2_MAX_ROWS_AFFECTS_ALL
- SQL_CA2_CRC_EXACT
- SQL_CA2_CRC_APPROXIMATE
- SQL_CA2_SIMULATE_NON_UNIQUE
- SQL_CA2_SIMULATE_TRY_UNIQUE
- SQL_CA2_SIMULATE_UNIQUE

SQL_KEYWORDS (string)
Indicates a comma-separated list of all data source-specific keywords. This character string is a list of all reserved keywords. Interoperable applications should not use these keywords in object names. This list does not contain keywords specific to ODBC or keywords that are used by both the data source and ODBC.

SQL_LIKE_ESCAPE_CLAUSE (string)
Indicates whether the data source supports an escape character for the percent character (%) and underscore (_) character in a LIKE predicate. Also, it indicates that the driver supports the ODBC syntax for defining a LIKE predicate escape character.

- "Y" indicates that there is support for escape characters in a LIKE predicate.
- "N" indicates that there is no support for escape characters in a LIKE predicate.

SQL_LOCK_TYPES (32-bit mask)
Reserved option, zero is returned for the bit-mask.

SQL_MAX_ASYNC_CONCURRENT_STATEMENTS (32-bit unsigned integer)
The maximum number of active concurrent statements in asynchronous mode that CLI can support on a given connection. This value is zero if there is no specific limit, or the limit is unknown.

SQL_MAX_BINARY_LITERAL_LEN (32-bit unsigned integer)
A 32-bit unsigned integer value specifying the maximum length (number of hexadecimal characters, excluding the literal prefix and suffix returned by `SQLGetTypeInfo()`) of a binary literal in an SQL statement. For example, the binary literal 0xFFAA has a length of 4. If there is no maximum length or the length is unknown, this value is set to zero.

SQL_MAX_CATALOG_NAME_LEN (16-bit integer)
The maximum length of a catalog name in the data source. This value is zero if there is no maximum length, or the length is unknown.

In previous versions of CLI this *fInfoType* argument was SQL_MAX_QUALIFIER_NAME_LEN.

SQL_MAX_CHAR_LITERAL_LEN (32-bit unsigned integer)
The maximum length of a character literal in an SQL statement (in bytes). Zero if there is no limit.

SQL_MAX_COLUMN_NAME_LEN (16-bit integer)
The maximum length of a column name (in bytes). Zero if there is no limit.

SQL_MAX_COLUMNS_IN_GROUP_BY (16-bit integer)
Indicates the maximum number of columns that the server supports in a GROUP BY clause. Zero if there is no limit.

SQL_MAX_COLUMNS_IN_INDEX (16-bit integer)
Indicates the maximum number of columns that the server supports in an index. Zero if there is no limit.

SQL_MAX_COLUMNS_IN_ORDER_BY (16-bit integer)
Indicates the maximum number of columns that the server supports in an ORDER BY clause. Zero if there is no limit.

SQL_MAX_COLUMNS_IN_SELECT (16-bit integer)
Indicates the maximum number of columns that the server supports in a SELECT list. Zero if there no limit.

SQL_MAX_COLUMNS_IN_TABLE (16-bit integer)
Indicates the maximum number of columns that the server supports in a base table. Zero if there is no limit.

SQL_MAX_CONCURRENT_ACTIVITIES (16-bit integer)
The maximum number of active environments that CLI can support. If there is no specified limit or the limit is unknown, this value is set to zero.

In previous versions of CLI this *InfoType* argument was SQL_ACTIVE_ENVIRONMENTS.

SQL_MAX_CURSOR_NAME_LEN (16-bit integer)
The maximum length of a cursor name (in bytes). This value is zero if there is no maximum length, or the length is unknown.

SQL_MAX_DRIVER_CONNECTIONS (16-bit integer)
The maximum number of active connections that are supported per application.

If the limit is dependent on system resources, zero is returned.

In previous versions of CLI this *InfoType* argument was SQL_ACTIVE_CONNECTIONS.

SQL_MAX_IDENTIFIER_LEN (16-bit integer)
The maximum size (in characters) that the data source supports for user-defined names.

SQL_MAX_INDEX_SIZE (32-bit unsigned integer)
Indicates the maximum size in bytes that the server supports for the combined columns in an index. Zero if no limit.

SQL_MAX_MODULE_NAME_LEN (16-bit integer)
Indicates the maximum length in bytes of a module qualifier name.

SQL_MAX_PROCEDURE_NAME_LEN (16-bit integer)
The maximum length of a procedure name (in bytes).

SQL_MAX_ROW_SIZE (32-bit unsigned integer)
Specifies the maximum length in bytes that the server supports in single row of a base table. Zero if there is no limit.

SQL_MAX_ROW_SIZE_INCLUDES_LONG (string)
Set to "Y" to indicate that the value that is returned by SQL_MAX_ROW_SIZE *InfoType* argument includes the length of product-specific *long string* data types. Otherwise, set to "N".

SQL_MAX_SCHEMA_NAME_LEN (16-bit integer)
The maximum length of a schema qualifier name (in bytes).

In previous versions of CLI this *fInfoType* argument was SQL_MAX_OWNER_NAME_LEN.

SQL_MAX_STATEMENT_LEN (32-bit unsigned integer)
Indicates the maximum length of an SQL statement string in bytes, including the number of white spaces in the statement.

SQL_MAX_TABLE_NAME_LEN (16-bit integer)
The maximum length of a table name (in bytes).

SQL_MAX_TABLES_IN_SELECT (16-bit integer)
Indicates the maximum number of table names in a FROM clause in a <query specification>.

SQL_MAX_USER_NAME_LEN (16-bit integer)
Indicates the maximum size for a <user identifier> (in bytes).

SQL_MODULE_USAGE (32-bit mask)
Indicates the type of SQL statements that have a module associated with them when these statements are executed. CLI always returns zero for DB2 for z/OS.

SQL_MU_PROCEDURE_INVOCATION is supported in the procedure invocation statement.

SQL_MULT_RESULT_SETS (string)
The character string "Y" indicates that the database supports multiple result sets, "N" indicates that it does not.

SQL_MULTIPLE_ACTIVE_TXN (string)
Indicates whether active transactions on multiple connections are permitted.

- "Y" indicates that multiple connections can have active transactions.
- "N" indicates that only one connection at a time can have an active transaction.CLI returns "N" for coordinated distributed unit of work (CONNECT TYPE 2) connections, (since the transaction or Unit Of Work spans all connections), and returns "Y" for all other connections.

SQL_NEED_LONG_DATA_LEN (string)
Indicates that a character string is reserved for the use of ODBC. “N” is always returned.

SQL_NON_NULLABLE_COLUMNS (16-bit integer)
Indicates whether non-nullable columns are supported. Valid returned values are:
- SQL_NNC_NON_NULL - Can be defined as NOT NULL.
- SQL_NNC_NULL - Cannot be defined as NOT NULL.

SQL_NULL_COLLATION (16-bit integer)
Indicates where NULLs are sorted in a result set. Valid returned values are:
- SQL_NC_HIGH - Null values sort high.
- SQL_NC_LOW - Null values sort low.

SQL_NUMERIC_FUNCTIONS (32-bit mask)
Indicates that the ODBC scalar numeric functions are supported. These functions are intended to be used with the ODBC vendor escape sequence.

The listed bit-masks are used to determine which numeric functions are supported:
- SQL_FN_NUM_ABS
- SQL_FN_NUM_ACOS
- SQL_FN_NUM_ASIN
- SQL_FN_NUM_ATAN
- SQL_FN_NUM_ATAN2
- SQL_FN_NUM_CEILING
- SQL_FN_NUM_COS
- SQL_FN_NUM_COT
- SQL_FN_NUM_DEGREES
- SQL_FN_NUM_EXP
- SQL_FN_NUM_FLOOR
- SQL_FN_NUM_LOG
- SQL_FN_NUM_LOG10
- SQL_FN_NUM_MOD
- SQL_FN_NUM_PI
- SQL_FN_NUM_POWER
- SQL_FN_NUM_RADIANS
- SQL_FN_NUM_RAND
- SQL_FN_NUM_ROUND
- SQL_FN_NUM_SIGN
- SQL_FN_NUM_SIN
- SQL_FN_NUM_SQRT
- SQL_FN_NUM_TAN
- SQL_FN_NUM_TRUNCATE

SQL_ODBC_API_CONFORMANCE (16-bit integer)
Indicates the level of ODBC conformance. Valid returned values are:
- SQL_OAC_NONE
- SQL_OAC_LEVEL1
- SQL_OAC_LEVEL2

SQL_ODBC_INTERFACE_CONFORMANCE (32-bit unsigned integer)
Indicates the level of the ODBC 3.0 interface that CLI conforms to:
- SQL_OIC_CORE : The minimum level that all ODBC drivers are expected to conform to. This level includes basic interface elements such as connection functions; functions for preparing and executing an SQL statement; basic result set metadata functions; basic catalog functions; and so on.
- SQL_OIC_LEVEL1 : A level that includes the core standards compliance level functionality, plus scrollable cursors, bookmarks, positioned updates and deletes; and so on.

- SQL_OIC_LEVEL2 : A level that includes the level 1 standards compliance level functionality, plus advanced features such as sensitive cursors; update, delete, and refresh by bookmarks; stored procedure support; catalog functions for primary and foreign keys; multi-catalog support; and so on.

SQL_ODBC_SAG_CLI_CONFORMANCE (16-bit integer)
The compliance to the functions of the SQL Access Group (SAG) CLI specification.

Valid returned values are:
- SQL_OSCC_NOT_COMPLIANT : The driver is not SAG-compliant.
- SQL_OSCC_COMPLIANT : The driver is SAG-compliant.

SQL_ODBC_SQL_CONFORMANCE (16-bit integer)
Valid returned values are:
- SQL_OSC_MINIMUM : Minimum ODBC SQL grammar supported
- SQL_OSC_CORE : Core ODBC SQL Grammar supported
- SQL_OSC_EXTENDED : Extended ODBC SQL Grammar supported

SQL_ODBC_VER (string)
The ODBC version number that the driver manager supports.

CLI returns the string “03.01.0000”. CLI returns the string "03.01.0000". For Windows 7 and Windows Server 2008 R2 operating systems, CLI returns the string “03.80.0000”.

SQL_OJ_CAPABILITIES (32-bit mask)
A 32-bit bit-mask enumerating the types of outer join supported.

The bitmasks are:
- SQL_OJ_LEFT : Left outer join is supported.
- SQL_OJ_RIGHT : Right outer join is supported.
- SQL_OJ_FULL : Full outer join is supported.
- SQL_OJ_NESTED : Nested outer join is supported.
- SQL_OJ_ORDERED : The order of the tables underlying the columns in the outer join ON clause do not have to be in the same order as the tables in the JOIN clause.
- SQL_OJ_INNER : The inner table of an outer join can also be an inner join.
- SQL_OJ_ALL_COMPARISONS_OPS : Any predicate can be used in the outer join ON clause. If this bit is not set, only the equality (=) comparison operator can be used in outer joins.

SQL_ORDER_BY_COLUMNS_IN_SELECT (string)
Set to "Y" if columns in the ORDER BY clauses must be in the select list; otherwise set to "N".

SQL_OUTER_JOINS (string)
The character string:
- "Y" indicates that outer joins are supported, and CLI supports the ODBC outer join request syntax.
- "N" indicates tha touter joins are not supported.

SQL_PARAM_ARRAY_ROW_COUNTS (32-bit unsigned integer)
Indicates the availability of row counts in a parameterized execution:
- SQL_PARC_BATCH, Individual row counts are available for each set of parameters. This behavior is conceptually equivalent to CLI generating a batch of SQL statements, one for each parameter set in the array. Extended error information can be retrieved by using the SQL_PARAM_STATUS_PTR descriptor field. To enable this behavior for

non-atomic operations, set the SQL_ATTR_PARC_BATCH connection attribute to SQL_PARC_BATCH_ENABLE and SQL_ATTR_PARAMOPT_ATOMIC to SQL_ATOMIC_NO. If SQL_ATTR_PARAMOPT_ATOMIC is set to SQL_ATOMIC_YES, the CLI0150E error message is returned.
- SQL_PARC_NO_BATCH : Only one row count is available, which is the cumulative row count resulting from the execution of the statement for the entire array of parameters. This behavior is conceptually equivalent to treating the statement along with the entire parameter array as one atomic unit. Errors are handled the same as if one statement was issued.

SQL_PARAM_ARRAY_SELECTS (32-bit unsigned integer)
Indicates the availability of result sets in a parameterized execution. Valid returned values are:
- SQL_PAS_BATCH : One result set is available per set of parameters. The SQL_PAS_BATCH is conceptually equivalent to CLI generating a batch of SQL statements, one for each parameter set in the array.
- SQL_PAS_NO_BATCH : Only one result set is available, which represents the cumulative result set resulting from the execution of the statement for the entire array of parameters. The SQL_PAS_NO_BATCH is conceptually equivalent to treating the statement along with the entire parameter array as one atomic unit.
- SQL_PAS_NO_SELECT : CLI does not allow a result-set generating statement to be executed with an array of parameters.

SQL_POS_OPERATIONS (32-bit mask)
Reserved option, zero is returned for the bit-mask.

SQL_POSITIONED_STATEMENTS (32-bit mask)
Indicates the degree of support for positioned UPDATE and positioned DELETE statements:
- SQL_PS_POSITIONED_DELETE
- SQL_PS_POSITIONED_UPDATE
- SQL_PS_SELECT_FOR_UPDATE - Indicates whether the server requires the FOR UPDATE clause to be specified on a <query expression> in order for a column to be updateable by using a cursor.

SQL_PROCEDURE_TERM (string)
The name a database vendor uses for a procedure

SQL_PROCEDURES (string)
A character string of "Y" indicates that the data source supports procedures and CLI supports the ODBC procedure invocation syntax specified by the CALL statement. "N" indicates that it does not.

SQL_QUOTED_IDENTIFIER_CASE (16-bit integer)
Valid returned values are:
- SQL_IC_UPPER : Not case sensitive and are stored in uppercase.
- SQL_IC_LOWER : Not case sensitive and are stored in lowercase.
- SQL_IC_SENSITIVE : Quoted identifiers (delimited identifiers) in SQL are case sensitive and are stored in mixed case in the system catalog.
- SQL_IC_MIXED - Not case sensitive and are stored in mixed case.

The SQL_QUOTED_ IDENTIFIER_CASE integer should be contrasted with the SQL_IDENTIFIER_CASE *InfoType* argument, which is used to determine how (unquoted) identifiers are stored in the system catalog.

SQL_ROW_UPDATES (string)
A character string of "Y" indicates a keyset-driven cursor or mixed cursor that maintains row versions or values for all fetched rows, and therefore

can detect any updates made to a row since the row was last fetched. This character string only applies to updates, not to deletions or insertions. CLI can return the SQL_ROW_UPDATED flag to the row status array when `SQLFetchScroll()` is called. Otherwise, "N" is returned.

SQL_SCHEMA_TERM (string)
The terminology of the database vendor for a schema (owner).

In previous versions of CLI this *InfoType* was SQL_OWNER_TERM.

SQL_SCHEMA_USAGE (32-bit mask)
Indicates the type of SQL statements that have schema (owners) associated with them when these statements are executed. Valid returned schema qualifiers (owners) are:
- SQL_SU_DML_STATEMENTS - All DML statements.
- SQL_SU_PROCEDURE_INVOCATION - The procedure invocation statement.
- SQL_SU_TABLE_DEFINITION - All table definition statements.
- SQL_SU_INDEX_DEFINITION - All index definition statements.
- SQL_SU_PRIVILEGE_DEFINITION - All privilege definition statements (grant and revoke statements).

In previous versions of CLI this *InfoType* argument was SQL_OWNER_USAGE.

SQL_SCROLL_CONCURRENCY (32-bit mask)
Indicates the concurrency options that are supported for the cursor.

The listed bit masks are used with the flag to determine which options are supported:
- SQL_SCCO_LOCK
- SQL_SCCO_READ_ONLY
- SQL_SCCO_TIMESTAMP
- SQL_SCCO_VALUES

CLI returns SQL_SCCO_LOCK, which indicates that the lowest level of locking that is sufficient to make an update.

SQL_SCROLL_OPTIONS (32-bit mask)
Indicates the scroll options that are supported for scrollable cursors.

The listed bit masks are used with the flag to determine which options are supported:
- SQL_SO_FORWARD_ONLY : The cursor scrolls only forward.
- SQL_SO_KEYSET_DRIVEN : CLI saves and uses the keys for every row in the result set.
- SQL_SO_STATIC : The data in the result set is static.
- SQL_SO_DYNAMIC : CLI keeps the keys for every row in the rowset (the keyset size is the same as the rowset size).
- SQL_SO_MIXED: CLI keeps the keys for every row in the keyset, and the keyset size is greater than the rowset size. The cursor is keyset-driven inside the keyset and dynamic outside the keyset.

SQL_SEARCH_PATTERN_ESCAPE (string)
Used to specify what the driver supports as an escape character for catalog functions, such as the `SQLTables()` function, and the `SQLColumns()` function.

SQL_SERVER_NAME (string)
Indicates the name of the DB2 instance. In contrast to the SQL_DATA_SOURCE_NAME character string, this character string is the

actual name of the database server. Some DBMSs provide a different name upon establishing a connection than the real server-name of the database.

SQL_SPECIAL_CHARACTERS (string)
A character string that contains only special characters (all characters except a...z, A...Z, 0...9, and underscore) that can be used in an identifier name, such as table, column, or index name, on the data source. For example, "@#". If an identifier contains special characters, the identifier must be a delimited identifier.

SQL_SQL_CONFORMANCE (32-bit unsigned integer)
Indicates the level of SQL-92 that is supported:
- SQL_SC_SQL92_ENTRY : Entry level SQL-92 compliant.
- SQL_SC_FIPS127_2_TRANSITIONAL : FIPS 127-2 transitional-level compliant.
- SQL_SC_SQL92_FULL : Full-level SQL-92 compliant.
- SQL_SC_ SQL92_INTERMEDIATE : Intermediate level SQL-92 compliant.

SQL_SQL92_DATETIME_FUNCTIONS (32-bit mask)
Indicates the datetime scalar functions that are supported by CLI and the data source. Valid returned values are:
- SQL_SDF_CURRENT_DATE
- SQL_SDF_CURRENT_TIME
- SQL_SDF_CURRENT_TIMESTAMP

SQL_SQL92_FOREIGN_KEY_DELETE_RULE (32-bit mask)
Indicates the rules that are supported for a foreign key in a DELETE statement, as defined by SQL-92. Valid returned values are:
- SQL_SFKD_CASCADE
- SQL_SFKD_NO_ACTION
- SQL_SFKD_SET_DEFAULT
- SQL_SFKD_SET_NULL

SQL_SQL92_FOREIGN_KEY_UPDATE_RULE (32-bit mask)
Indicates the rules that are supported for a foreign key in an UPDATE statement, as defined by SQL-92. Valid returned values are:
- SQL_SFKU_CASCADE
- SQL_SFKU_NO_ACTION
- SQL_SFKU_SET_DEFAULT
- SQL_SFKU_SET_NULL

SQL_SQL92_GRANT (32-bit mask)
Indicates the clauses that are supported in a GRANT statement, as defined by SQL-92. Valid returned values are:
- SQL_SG_DELETE_TABLE
- SQL_SG_INSERT_COLUMN
- SQL_SG_INSERT_TABLE
- SQL_SG_REFERENCES_TABLE
- SQL_SG_REFERENCES_COLUMN
- SQL_SG_SELECT_TABLE
- SQL_SG_UPDATE_COLUMN
- SQL_SG_UPDATE_TABLE
- SQL_SG_USAGE_ON_DOMAIN
- SQL_SG_USAGE_ON_CHARACTER_SET
- SQL_SG_USAGE_ON_COLLATION
- SQL_SG_USAGE_ON_TRANSLATION
- SQL_SG_WITH_GRANT_OPTION

SQL_SQL92_NUMERIC_VALUE_FUNCTIONS (32-bit mask)
Indicates the numeric value scalar functions that are supported by CLI and the data source, as defined in SQL-92. Valid returned values are:
- SQL_SNVF_BIT_LENGTH
- SQL_SNVF_CHAR_LENGTH
- SQL_SNVF_CHARACTER_LENGTH
- SQL_SNVF_EXTRACT
- SQL_SNVF_OCTET_LENGTH
- SQL_SNVF_POSITION

SQL_SQL92_PREDICATES (32-bit mask)
Indicates the predicates that are supported in a SELECT statement, as defined by SQL-92. Valid returned values are:
- SQL_SP_BETWEEN
- SQL_SP_COMPARISON
- SQL_SP_EXISTS
- SQL_SP_IN
- SQL_SP_ISNOTNULL
- SQL_SP_ISNULL
- SQL_SP_LIKE
- SQL_SP_MATCH_FULL
- SQL_SP_MATCH_PARTIAL
- SQL_SP_MATCH_UNIQUE_FULL
- SQL_SP_MATCH_UNIQUE_PARTIAL
- SQL_SP_OVERLAPS
- SQL_SP_QUANTIFIED_COMPARISON
- SQL_SP_UNIQUE

SQL_SQL92_RELATIONAL_JOIN_OPERATORS (32-bit mask)
Indicates the relational join operators that are supported in a SELECT statement, as defined by SQL-92. Valid returned values are:
- SQL_SRJO_CORRESPONDING_CLAUSE
- SQL_SRJO_CROSS_JOIN
- SQL_SRJO_EXCEPT_JOIN
- SQL_SRJO_FULL_OUTER_JOIN
- SQL_SRJO_INNER_JOIN (indicates support for the INNER JOIN syntax, not for the inner join capability)
- SQL_SRJO_INTERSECT_JOIN
- SQL_SRJO_LEFT_OUTER_JOIN
- SQL_SRJO_NATURAL_JOIN
- SQL_SRJO_RIGHT_OUTER_JOIN
- SQL_SRJO_UNION_JOIN

SQL_SQL92_REVOKE (32-bit mask)
Indicates which clauses the data source supports in the REVOKE statement, as defined by SQL-92. Valid returned values are:
- SQL_SR_CASCADE
- SQL_SR_DELETE_TABLE
- SQL_SR_GRANT_OPTION_FOR
- SQL_SR_INSERT_COLUMN
- SQL_SR_INSERT_TABLE
- SQL_SR_REFERENCES_COLUMN
- SQL_SR_REFERENCES_TABLE
- SQL_SR_RESTRICT
- SQL_SR_SELECT_TABLE
- SQL_SR_UPDATE_COLUMN
- SQL_SR_UPDATE_TABLE

- SQL_SR_USAGE_ON_DOMAIN
- SQL_SR_USAGE_ON_CHARACTER_SET
- SQL_SR_USAGE_ON_COLLATION
- SQL_SR_USAGE_ON_TRANSLATION

SQL_SQL92_ROW_VALUE_CONSTRUCTOR (32-bit mask)
Indicates the row value constructor expressions that are supported in a SELECT statement, as defined by SQL-92. Valid returned values are:
- SQL_SRVC_DEFAULT
- SQL_SRVC_NULL
- SQL_SRVC_ROW_SUBQUERY
- SQL_SRVC_VALUE_EXPRESSION

SQL_SQL92_STRING_FUNCTIONS (32-bit mask)
Indicates the string scalar functions that are supported by CLI and the data source, as defined by SQL-92. Valid returned values are:
- SQL_SSF_CONVERT
- SQL_SSF_LOWER
- SQL_SSF_SUBSTRING
- SQL_SSF_TRANSLATE
- SQL_SSF_TRIM_BOTH
- SQL_SSF_TRIM_LEADING
- SQL_SSF_TRIM_TRAILING
- SQL_SSF_UPPER

SQL_SQL92_VALUE_EXPRESSIONS (32-bit mask)
Indicates the value expressions that are supported, as defined by SQL-92. Valid returned values are:
- SQL_SVE_CASE
- SQL_SVE_CAST
- SQL_SVE_COALESCE
- SQL_SVE_NULLIF

SQL_STANDARD_CLI_CONFORMANCE (32-bit mask)
Indicates the CLI standard or standards to which CLI conforms. Valid returned values are:
- SQL_SCC_ISO92_CLI
- SQL_SCC_XOPEN_CLI_VERSION1

SQL_STATIC_CURSOR_ATTRIBUTES1 (32-bit mask)
Indicates the attributes of a static cursor that are supported by CLI. Valid returned values are (subset 1 of 2):
- SQL_CA1_ABSOLUTE
- SQL_CA1_BOOKMARK
- SQL_CA1_BULK_ADD
- SQL_CA1_BULK_DELETE_BY_BOOKMARK
- SQL_CA1_BULK_FETCH_BY_BOOKMARK
- SQL_CA1_BULK_UPDATE_BY_BOOKMARK
- SQL_CA1_LOCK_EXCLUSIVE
- SQL_CA1_LOCK_NO_CHANGE
- SQL_CA1_LOCK_UNLOCK
- SQL_CA1_NEXT
- SQL_CA1_POS_DELETE
- SQL_CA1_POS_POSITION
- SQL_CA1_POS_REFRESH
- SQL_CA1_POS_UPDATE
- SQL_CA1_POSITIONED_UPDATE
- SQL_CA1_POSITIONED_DELETE

- SQL_CA1_RELATIVE
- SQL_CA1_SELECT_FOR_UPDATE

SQL_STATIC_CURSOR_ATTRIBUTES2 (32-bit mask)
Indicates the attributes of a static cursor that are supported by CLI (subset 2 of 2):
- SQL_CA2_READ_ONLY_CONCURRENCY
- SQL_CA2_LOCK_CONCURRENCY
- SQL_CA2_OPT_ROWVER_CONCURRENCY
- SQL_CA2_OPT_VALUES_CONCURRENCY
- SQL_CA2_SENSITIVITY_ADDITIONS
- SQL_CA2_SENSITIVITY_DELETIONS
- SQL_CA2_SENSITIVITY_UPDATES
- SQL_CA2_MAX_ROWS_SELECT
- SQL_CA2_MAX_ROWS_INSERT
- SQL_CA2_MAX_ROWS_DELETE
- SQL_CA2_MAX_ROWS_UPDATE
- SQL_CA2_MAX_ROWS_CATALOG
- SQL_CA2_MAX_ROWS_AFFECTS_ALL
- SQL_CA2_CRC_EXACT
- SQL_CA2_CRC_APPROXIMATE
- SQL_CA2_SIMULATE_NON_UNIQUE
- SQL_CA2_SIMULATE_TRY_UNIQUE
- SQL_CA2_SIMULATE_UNIQUE

SQL_STATIC_SENSITIVITY (32-bit mask)
Indicates whether changes that are made by an application with a positioned update or delete statement can be detected by that application. Valid returned values are:
- SQL_SS_ADDITIONS : Added rows are visible to the cursor, and the cursor can scroll to these rows. All DB2 servers see added rows.
- SQL_SS_DELETIONS : Deleted rows are no longer available to the cursor, and do not leave a hole in the result set. After the cursor scrolls from a deleted row, it cannot return to that row.
- SQL_SS_UPDATES : Updated rows are visible to the cursor. If the cursor scrolls from and returns to an updated row, the data that is returned by the cursor is the updated data, not the original data.

SQL_STRING_FUNCTIONS (32-bit mask)
Indicates which string functions are supported.

The listed bit masks are used to determine which string functions are supported:
- SQL_FN_STR_ASCII
- SQL_FN_STR_BIT_LENGTH
- SQL_FN_STR_CHAR
- SQL_FN_STR_CHAR_LENGTH
- SQL_FN_STR_CHARACTER_LENGTH
- SQL_FN_STR_CONCAT
- SQL_FN_STR_DIFFERENCE
- SQL_FN_STR_INSERT
- SQL_FN_STR_LCASE
- SQL_FN_STR_LEFT
- SQL_FN_STR_LENGTH
- SQL_FN_STR_LOCATE
- SQL_FN_STR_LOCATE_2
- SQL_FN_STR_LTRIM
- SQL_FN_STR_OCTET_LENGTH

- SQL_FN_STR_POSITION
- SQL_FN_STR_REPEAT
- SQL_FN_STR_REPLACE
- SQL_FN_STR_RIGHT
- SQL_FN_STR_RTRIM
- SQL_FN_STR_SOUNDEX
- SQL_FN_STR_SPACE
- SQL_FN_STR_SUBSTRING
- SQL_FN_STR_UCASE

If an application can call the LOCATE scalar function with the *string_exp1, string_exp2,* and *start* arguments, the SQL_FN_STR_LOCATE bit mask is returned. If an application can call the LOCATE scalar function only with the *string_exp1* and *string_exp2*, the SQL_FN_STR_LOCATE_2 bit mask is returned. If the LOCATE scalar function is fully supported, both bit masks are returned.

SQL_SUBQUERIES (32-bit mask)
Indicates which predicates support subqueries. Valid returned values are:
- SQL_SQ_COMPARISION : The *comparison* predicate.
- SQL_SQ_CORRELATE_SUBQUERIES : All predicates that support subqueries also support correlated subqueries.
- SQL_SQ_EXISTS : The *exists* predicate.
- SQL_SQ_IN : The *in* predicate.
- SQL_SQ_QUANTIFIED : The predicates that contains a quantification scalar function.

SQL_SYSTEM_FUNCTIONS (32-bit mask)
Indicates which scalar system functions are supported.

The listed bit masks are used to determine which scalar system functions are supported:
- SQL_FN_SYS_DBNAME
- SQL_FN_SYS_IFNULL
- SQL_FN_SYS_USERNAME

Note: These functions are intended to be used with the escape sequence in ODBC.

SQL_TABLE_TERM (string)
The terminology of a database vendor for a table.

SQL_TIMEDATE_ADD_INTERVALS (32-bit mask)
Indicates whether or not the special ODBC system function TIMESTAMPADD is supported, and, if it is, which intervals are supported.

The listed bit masks are used to determine which intervals are supported:
- SQL_FN_TSI_FRAC_SECOND
- SQL_FN_TSI_SECOND
- SQL_FN_TSI_MINUTE
- SQL_FN_TSI_HOUR
- SQL_FN_TSI_DAY
- SQL_FN_TSI_WEEK
- SQL_FN_TSI_MONTH
- SQL_FN_TSI_QUARTER
- SQL_FN_TSI_YEAR

SQL_TIMEDATE_DIFF_INTERVALS (32-bit mask)
Indicates whether or not the special ODBC system function TIMESTAMPDIFF is supported, and, if it is, which intervals are supported.

The listed bit masks are used to determine which intervals are supported:
- SQL_FN_TSI_FRAC_SECOND
- SQL_FN_TSI_SECOND
- SQL_FN_TSI_MINUTE
- SQL_FN_TSI_HOUR
- SQL_FN_TSI_DAY
- SQL_FN_TSI_WEEK
- SQL_FN_TSI_MONTH
- SQL_FN_TSI_QUARTER
- SQL_FN_TSI_YEAR

SQL_TIMEDATE_FUNCTIONS (32-bit mask)
Indicates which time and date functions are supported.

The listed bit masks are used to determine which date functions are supported:
- SQL_FN_TD_CURRENT_DATE
- SQL_FN_TD_CURRENT_TIME
- SQL_FN_TD_CURRENT_TIMESTAMP
- SQL_FN_TD_CURDATE
- SQL_FN_TD_CURTIME
- SQL_FN_TD_DAYNAME
- SQL_FN_TD_DAYOFMONTH
- SQL_FN_TD_DAYOFWEEK
- SQL_FN_TD_DAYOFYEAR
- SQL_FN_TD_EXTRACT
- SQL_FN_TD_HOUR
- SQL_FN_TD_JULIAN_DAY
- SQL_FN_TD_MINUTE
- SQL_FN_TD_MONTH
- SQL_FN_TD_MONTHNAME
- SQL_FN_TD_NOW
- SQL_FN_TD_QUARTER
- SQL_FN_TD_SECOND
- SQL_FN_TD_SECONDS_SINCE_MIDNIGHT
- SQL_FN_TD_TIMESTAMPADD
- SQL_FN_TD_TIMESTAMPDIFF
- SQL_FN_TD_WEEK
- SQL_FN_TD_YEAR

Note: These functions are intended to be used with the escape sequence in ODBC.

SQL_TXN_CAPABLE (16-bit integer)
Indicates whether transactions can contain DDL, DML, or both. Valid returned values are:
- SQL_TC_NONE : Transactions not supported.
- SQL_TC_DML : Transactions can contain only DML statements (for example, SELECT, INSERT, UPDATE and DELETE). DDL statements, such as CREATE TABLE and DROP INDEX, that are encountered in a transaction cause an error.
- SQL_TC_DDL_COMMIT : Transactions can only contain DML statements. DDL statements that are encountered in a transaction cause the transaction to be committed.
- SQL_TC_DDL_IGNORE : Transactions can only contain DML statements. DDL statements that are encountered in a transaction are ignored.

- SQL_TC_ALL : Transactions can contain DDL and DML statements in any order.

SQL_TXN_ISOLATION_OPTION (32-bit mask)
The transaction isolation levels that are available at the currently connected database server.

The listed masks are used in conjunction with the flag to determine which options are supported:
- SQL_TXN_READ_UNCOMMITTED
- SQL_TXN_READ_COMMITTED
- SQL_TXN_REPEATABLE_READ
- SQL_TXN_SERIALIZABLE
- SQL_TXN_NOCOMMIT
- SQL_TXN_VERSIONING

For descriptions of each level, see SQL_DEFAULT_TXN_ISOLATION.

SQL_UNION (32-bit mask)
Indicates if the server supports the UNION operator. Valid returned values are:
- SQL_U_UNION : Supports the UNION clause.
- SQL_U_UNION_ALL : Supports the ALL keyword in the UNION clause.

If SQL_U_UNION_ALL is set, so is SQL_U_UNION.

SQL_USER_NAME (string)
Indicates the user name that is used in a particular database. This character string is the identifier that is specified on the `SQLConnect()` call.

SQL_XOPEN_CLI_YEAR (string)
Indicates the year of publication of the X/Open specification with which the version of the driver fully complies.

SQLGetLength function (CLI) - Retrieve length of a string value

Retrieves the length of a large object value, referenced by a large object locator that has been returned from the server (as a result of a fetch, or an SQLGetSubString() call) during the current transaction.

Specification:

- **CLI** 2.1

Syntax

```
SQLRETURN  SQLGetLength   (SQLHSTMT          StatementHandle,  /* hstmt */
                           SQLSMALLINT       LocatorCType,
                           SQLINTEGER        Locator,
                           SQLINTEGER        *StringLength,
                           SQLINTEGER        *IndicatorValue);
```

Function arguments

Table 95. SQLGetLength arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle. This can be any statement handle which has been allocated but which does not currently have a prepared statement assigned to it.

Table 95. SQLGetLength arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*LocatorCType*	input	The C type of the source LOB locator. This may be: • SQL_C_BLOB_LOCATOR • SQL_C_CLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR
SQLINTEGER	*Locator*	input	Must be set to the LOB locator value.
SQLINTEGER *	*StringLength*	output	The length of the returned information in *rgbValue* in bytes[a] if the target C buffer type is intended for a binary or character string variable and not a locator value. If the pointer is set to NULL then the SQLSTATE **HY**009 is returned.
SQLINTEGER *	*IndicatorValue*	output	Always set to zero.

Note:

a This is in characters for DBCLOB data.

Usage

SQLGetLength() can be used to determine the length of the data value represented by a LOB locator. It is used by applications to determine the overall length of the referenced LOB value so that the appropriate strategy to obtain some or all of the LOB value can be chosen. The length is calculated by the database server using the server code page, and so if the application code page is different from the server code page, then there may be some complexity in calculating space requirements on the client. The application will need to allow for code page expansion if any is needed.

The *Locator* argument can contain any valid LOB locator which has not been explicitly freed using a FREE LOCATOR statement nor implicitly freed because the transaction during which it was created has ended.

The statement handle must not have been associated with any prepared statements or catalog function calls.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 96. SQLGetLength SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The combination of *LocatorCType* and *Locator* is not valid.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.

Table 96. SQLGetLength SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY003	Program type out of range.	*LocatorCType* is not one of SQL_C_CLOB_LOCATOR, SQL_C_BLOB_LOCATOR, or SQL_C_DBCLOB_LOCATOR.
HY009	Invalid argument value.	Pointer to *StringLength* was NULL.
HY010	Function sequence error.	The specified *StatementHandle* is not in an *allocated* state. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.
0F001	The LOB token variable does not currently represent any value.	The value specified for *Locator* has not been associated with a LOB locator.

Restrictions

This function is not available when connected to a DB2 server that does not support large objects. Call SQLGetFunctions() with the function type set to SQL_API_SQLGETLENGTH and check the *fExists* output argument to determine if the function is supported for the current connection.

Example

```
/* get the length of the whole CLOB data */
cliRC = SQLGetLength(hstmtLocUse,
                     SQL_C_CLOB_LOCATOR,
                     clobLoc,
                     &clobLen,
                     &ind);
```

SQLGetPosition function (CLI) - Return starting position of string

Returns the starting position of one string within a LOB value (the source).

The source value must be a LOB locator, the search string can be a LOB locator or a literal string.

Specification:

- **CLI** 2.1

The source and search LOB locators can be any that have been returned from the database from a fetch or a SQLGetSubString() call during the current transaction.

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLGetPositionW(). For information about ANSI to Unicode function mappings, refer to "Unicode functions (CLI)" on page 5.

Syntax

```
SQLRETURN   SQLGetPosition   (SQLHSTMT          StatementHandle,   /* hstmt */
                              SQLSMALLINT       LocatorCType,
                              SQLINTEGER        SourceLocator,
                              SQLINTEGER        SearchLocator,
                              SQLCHAR           *SearchLiteral,
                              SQLINTEGER        SearchLiteralLength,
                              SQLUINTEGER       FromPosition,
                              SQLUINTEGER       *LocatedAt,
                              SQLINTEGER        *IndicatorValue);
```

Function arguments

Table 97. SQLGetPosition arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle. This can be any statement handle which has been allocated but which does not currently have a prepared statement assigned to it.
SQLSMALLINT	*LocatorCType*	input	The C type of the source LOB locator. This can be: • SQL_C_BLOB_LOCATOR • SQL_C_CLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR
SQLINTEGER	*Locator*	input	*Locator* must be set to the source LOB locator.
SQLINTEGER	*SearchLocator*	input	If the *SearchLiteral* pointer is NULL and if *SearchLiteralLength* is set to 0, then *SearchLocator* must be set to the LOB locator associated with the search string; otherwise, this argument is ignored.
SQLCHAR *	*SearchLiteral*	input	This argument points to the area of storage that contains the search string literal. If *SearchLiteralLength* is 0, this pointer must be NULL.
SQLINTEGER	*SearchLiteralLength*	input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store *SearchLiteral*(in bytes). [a] If this argument value is 0, then the argument *SearchLocator* is meaningful.
SQLUINTEGER	*FromPosition*	input	For BLOBs and CLOBs, this is the position of the first byte within the source string at which the search is to start. For DBCLOBs, this is the first character. The start byte or character is numbered 1.
SQLUINTEGER *	*LocatedAt*	output	For BLOBs and CLOBs, this is the byte position at which the string was located or, if not located, the value zero. For DBCLOBs, this is the character position. If the length of the source string is zero, the value 1 is returned.
SQLINTEGER *	*IndicatorValue*	output	Always set to zero.

Note:

a This is in bytes even for DBCLOB data.

Usage

SQLGetPosition() is used in conjunction with SQLGetSubString() in order to obtain any portion of a LOB in a random manner. In order to use SQLGetSubString(), the location of the substring within the overall string must be known in advance. In situations where the start of that substring can be found by a search string, SQLGetPosition() can be used to obtain the starting position of that substring.

The *Locator* and *SearchLocator* (if used) arguments can contain any valid LOB locator which has not been explicitly freed using a FREE LOCATOR statement or implicitly freed because the transaction during which it was created has ended.

The *Locator* and *SearchLocator* must have the same LOB locator type.

The statement handle must not have been associated with any prepared statements or catalog function calls.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 98. SQLGetPosition SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The combination of *LocatorCType* and either of the LOB locator values is not valid.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY009	Invalid argument value.	The pointer to the *LocatedAt* argument was NULL. The argument value for *FromPosition* was not greater than 0. *LocatorCType* is not one of SQL_C_CLOB_LOCATOR, SQL_C_BLOB_LOCATOR, or SQL_C_DBCLOB_LOCATOR.

Table 98. SQLGetPosition SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The specified *StatementHandle* is not in an *allocated* state. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value of *SearchLiteralLength* was less than 1, and not SQL_NTS. The length of the pattern is longer than the maximum data length of the associated variable SQL data type (for DB2 for z/OS servers, the pattern length is a maximum of 4000 bytes regardless of the data type or the *LocatorCType*). For LocatorCType of SQL_C_CLOB_LOCATOR, the literal maximum size is that of an SQLCLOB; for LocatorCType of SQL_C_BLOB_LOCATOR, the literal maximum size is that of an SQLVARBINARY; for LocatorCType of SQL_C_DBCLOB_LOCATOR, the literal maximum size is that of an SQLVARGRAPHIC.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.
0F001	The LOB token variable does not currently represent any value.	The value specified for *Locator* or *SearchLocator* is not currently a LOB locator.

Restrictions

This function is not available when connected to a DB2 server that does not support large objects. Call SQLGetFunctions() with the function type set to SQL_API_SQLGETPOSITION and check the *fExists* output argument to determine if the function is supported for the current connection.

The SQLGetPosition() function is intended to handle graphic data and not WCHAR data. As a result, the data passed to this function should be considered big endian.

Example

```
/* get the starting position of the CLOB piece of data */
cliRC = SQLGetPosition(hstmtLocUse,
                       SQL_C_CLOB_LOCATOR,
                       clobLoc,
                       0,
                       (SQLCHAR *)"Interests",
                       strlen("Interests"),
                       1,
                       &clobPiecePos,
                       &ind);
```

SQLGetSQLCA function (CLI) - Get SQLCA data structure

SQLGetSQLCA() has been deprecated. Use `SQLGetDiagField()` and `SQLGetDiagRec()` to retrieve diagnostic information.

Although this version of CLI continues to support `SQLGetSQLCA()`, it is recommended that you stop using it in your CLI programs so that they conform to the latest standards.

SQLGetStmtAttr function (CLI) - Get current setting of a statement attribute

Returns the current setting of a statement attribute.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLGetStmtAttrW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLGetStmtAttr   (SQLHSTMT          StatementHandle,
                              SQLINTEGER        Attribute,
                              SQLPOINTER        ValuePtr,
                              SQLINTEGER        BufferLength,
                              SQLINTEGER        *StringLengthPtr);
```

Function arguments

Table 99. SQLGetStmtAttr arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLINTEGER	*Attribute*	input	Attribute to retrieve.
SQLPOINTER	*ValuePtr*	output	Pointer to a buffer in which to return the value of the attribute specified in *Attribute*.

Table 99. SQLGetStmtAttr arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER	*BufferLength*	input	If *Attribute* is an ODBC-defined attribute and *ValuePtr* points to a character string or a binary buffer, this argument should be the length of **ValuePtr*. If *Attribute* is an ODBC-defined attribute and **ValuePtr* is an integer, *BufferLength* is ignored. If *Attribute* is a CLI attribute, the application indicates the nature of the attribute by setting the *BufferLength* argument. *BufferLength* can have the following values: • If **ValuePtr* is a pointer to a character string, then *BufferLength* is the number of bytes needed to store the string, or SQL_NTS. • If **ValuePtr* is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in *BufferLength*. This places a negative value in *BufferLength*. • If **ValuePtr* is a pointer to a value other than a character string or binary string, then *BufferLength* should have the value SQL_IS_POINTER. • If **ValuePtr* is contains a fixed-length data type, then *BufferLength* is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate. • If the value returned in *ValuePtr* is a Unicode string, the *BufferLength* argument must be an even number.
SQLSMALLINT *	*StringLengthPtr*	output	A pointer to a buffer in which to return the total number of bytes (excluding the null termination character) available to return in **ValuePtr*. If this is a null pointer, no length is returned. If the attribute value is a character string, and the number of bytes available to return is greater than or equal to *BufferLength*, the data in **ValuePtr* is truncated to *BufferLength* minus the length of a null termination character and is null-terminated by the CLI.

Usage

A call to SQLGetStmtAttr() returns in **ValuePtr* the value of the statement attribute specified in *Attribute*. That value can either be a 32-bit value or a null-terminated character string. If the value is a null-terminated string, the application specifies the maximum length of that string in the *BufferLength* argument, and CLI returns the length of that string in the **StringLengthPtr* buffer. If the value is a 32-bit value, the *BufferLength* and *StringLengthPtr* arguments are not used.

The following statement attributes are read-only, so can be retrieved by SQLGetStmtAttr(), but not set by SQLSetStmtAttr(). Refer to the list of statement attributes for all statement attributes that can be set and retrieved.
- SQL_ATTR_IMP_PARAM_DESC
- SQL_ATTR_IMP_ROW_DESC
- SQL_ATTR_ROW_NUMBER

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 100. SQLGetStmtAttr SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The data returned in **ValuePtr* was truncated to be *BufferLength* minus the length of a null termination character. The length of the untruncated string value is returned in **StringLengthPtr*. (Function returns SQL_SUCCESS_WITH_INFO.)
24000	Invalid cursor state.	The argument *Attribute* was SQL_ATTR_ROW_NUMBER and the cursor was not open, or the cursor was positioned before the start of the result set or after the end of the result set.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	An asynchronously executing function was called for the *StatementHandle* and was still executing when this function was called. SQLExecute() or SQLExecDirect() was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value specified for argument *BufferLength* was less than 0.
HY092	Option type out of range.	The value specified for the argument *Attribute* was not valid for this version of CLI
HY109	Invalid cursor position.	The *Attribute* argument was SQL_ATTR_ROW_NUMBER and the row had been deleted or could not be fetched.
HYC00	Driver not capable.	The value specified for the argument *Attribute* was a valid CLI attribute for the version of CLI, but was not supported by the data source.

Restrictions

None.

Example

```
/* get the handle for the implicitly allocated descriptor */
rc = SQLGetStmtAttr(hstmt,
                    SQL_ATTR_IMP_ROW_DESC,
                    &hIRD,
                    SQL_IS_INTEGER,
                    &indicator);
```

SQLGetStmtOption function (CLI) - Return current setting of a statement option

In ODBC 3.0, SQLGetStmtOption() has been deprecated and replaced with SQLGetStmtAttr().

Although this version of CLI continues to support SQLGetStmtOption(), use SQLGetStmtAttr() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

The statement:

```
SQLGetStmtOption(hstmt, SQL_ATTR_CURSOR_HOLD, pvCursorHold);
```

for example, would be rewritten using the new function as:

```
SQLGetStmtAttr(hstmt, SQL_ATTR_CURSOR_HOLD, pvCursorHold,
               SQL_IS_INTEGER, NULL);
```

SQLGetSubString function (CLI) - Retrieve portion of a string value

Retrieves a portion of a large object value, referenced by a large object locator that has been returned from the server (returned by a fetch or a previous SQLGetSubString() call) during the current transaction.

Specification:

- **CLI** 2.1

Syntax

```
SQLRETURN  SQLGetSubString   (
             SQLHSTMT          StatementHandle,   /* hstmt */
             SQLSMALLINT       LocatorCType,
             SQLINTEGER        SourceLocator,
             SQLUINTEGER       FromPosition,
             SQLUINTEGER       ForLength,
             SQLSMALLINT       TargetCType,
             SQLPOINTER        DataPtr,           /* rgbValue */
             SQLINTEGER        BufferLength,      /* cbValueMax */
             SQLINTEGER        *StringLength,
             SQLINTEGER        *IndicatorValue);
```

Function arguments

Table 101. SQLGetSubString arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle. This can be any statement handle which has been allocated but which does not currently have a prepared statement assigned to it.

Table 101. SQLGetSubString arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*LocatorCType*	input	The C type of the source LOB locator. This may be: • SQL_C_BLOB_LOCATOR • SQL_C_CLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR
SQLINTEGER	*Locator*	input	*Locator* must be set to the source LOB locator value.
SQLUINTEGER	*FromPosition*	input	For BLOBs and CLOBs, this is the position of the first byte to be returned by the function. For DBCLOBs, this is the first character. The start byte or character is numbered 1.
SQLUINTEGER	*ForLength*	input	This is the length of the string to be returned by the function. For BLOBs and CLOBs, this is the length in bytes. For DBCLOBs, this is the length in characters. If *FromPosition* is less than the length of the source string but *FromPosition* + *ForLength* - 1 extends beyond the end of the source string, the result is padded on the right with the necessary number of characters (X'00' for BLOBs, single byte blank character for CLOBs, and double byte blank character for DBCLOBs).
SQLSMALLINT	*TargetCType*	input	The C data type of the *DataPtr*. The target must always be either a LOB locator C buffer type: • SQL_C_CLOB_LOCATOR • SQL_C_BLOB_LOCATOR • SQL_C_DBCLOB_LOCATOR or a C string type: • SQL_C_CHAR • SQL_C_WCHAR • SQL_C_BINARY • SQL_C_DBCHAR
SQLPOINTER	*DataPtr*	output	Pointer to the buffer where the retrieved string value or a LOB locator is to be stored.
SQLINTEGER	*BufferLength*	input	Maximum size of the buffer pointed to by *DataPtr* in bytes.
SQLINTEGER *	*StringLength*	output	The length of the returned information in *DataPtr* in bytes[a] if the target C buffer type is intended for a binary or character string variable and not a locator value. If the pointer is set to NULL, nothing is returned.
SQLINTEGER *	*IndicatorValue*	output	Always set to zero.

Note:

a This is in bytes even for DBCLOB data.

Usage

SQLGetSubString() is used to obtain any portion of the string that is represented by the LOB locator. There are two choices for the target:

- The target can be an appropriate C string variable.

- A new LOB value can be created on the server and the LOB locator for that value can be assigned to a target application variable on the client.

SQLGetSubString() can be used as an alternative to SQLGetData() for getting LOB data in pieces. In this case a column is first bound to a LOB locator, which is then used to fetch the LOB as a whole or in pieces.

The Locator argument can contain any valid LOB locator which has not been explicitly freed using a FREE LOCATOR statement nor implicitly freed because the transaction during which it was created has ended.

The statement handle must not have been associated with any prepared statements or catalog function calls.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 102. SQLGetSubString SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The amount of data to be returned is longer than *BufferLength*. The actual length of data available for return is stored in *StringLength*.
07006	Invalid conversion.	The value specified for *TargetCType* was not SQL_C_CHAR, SQL_WCHAR, SQL_C_BINARY, SQL_C_DBCHAR, or a LOB locator. The value specified for *TargetCType* is inappropriate for the source (for example SQL_C_DBCHAR for a BLOB column).
22011	A substring error occurred.	*FromPosition* is greater than the of length of the source string.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY003	Program type out of range.	*LocatorCType* is not one of SQL_C_CLOB_LOCATOR, SQL_C_BLOB_LOCATOR, or SQL_C_DBCLOB_LOCATOR.
HY009	Invalid argument value.	The value specified for *FromPosition* or for *ForLength* was not a positive integer.

Table 102. SQLGetSubString SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The specified *StatementHandle* is not in an *allocated* state. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The value of *BufferLength* was less than 0.
HYC00	Driver not capable.	The application is currently connected to a data source that does not support large objects.
0F001	No locator currently assigned	The value specified for *Locator* is not currently a LOB locator.

Restrictions

This function is not available when connected to a DB2 server that does not support large objects. Call SQLGetFunctions() with the function type set to SQL_API_SQLGETSUBSTRING and check the *fExists* output argument to determine if the function is supported for the current connection.

FromPosition of SQLGetSubstring() can not take zero or negative values for IDS Data Servers. This is a current limitation.

When accessing IDS data servers, calling the SQLGetSubString() function with a TargetCType argument value of SQL_C_CLOB_LOCATOR or SQL_C_BLOB_LOCATOR will return an "Invalid Conversion" error. This conversion is not supported.

Example

```
/* read the piece of CLOB data in buffer */
cliRC = SQLGetSubString(hstmtLocUse,
                        SQL_C_CLOB_LOCATOR,
                        clobLoc,
                        clobPiecePos,
                        clobLen - clobPiecePos,
                        SQL_C_CHAR,
                        buffer,
                        clobLen - clobPiecePos + 1,
                        &clobPieceLen,
                        &ind);
```

SQLGetTypeInfo function (CLI) - Get data type information

Returns information about the data types that are supported by the DBMSs associated with CLI.

The information is returned in an SQL result set. The columns can be received using the same functions that are used to process a query.

SQLGetTypeInfo function (CLI) - Get data type information

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

Syntax

```
SQLRETURN   SQLGetTypeInfo (
               SQLHSTMT          StatementHandle,  /* hstmt */
               SQLSMALLINT       DataType);        /* fSqlType */
```

Function arguments

Table 103. SQLGetTypeInfo arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLSMALLINT	*DataType*	input	The SQL data type being queried. The supported types are: • SQL_ALL_TYPES • SQL_BIGINT • SQL_BINARY • SQL_BIT • SQL_BLOB • SQL_CHAR • SQL_CLOB • SQL_DATE [1] • SQL_TYPE_DATE • SQL_DBCLOB • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_SMALLINT • SQL_TIME [1] • SQL_TIMESTAMP [1] • SQL_TYPE_TIME • SQL_TYPE_TIMESTAMP • SQL_TINYINT • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC • SQL_XML If SQL_ALL_TYPES is specified, information about all supported data types would be returned in ascending order by TYPE_NAME. All unsupported data types would be absent from the result set.

Note:

1. These SQL data types are supported for compatibility with ODBC 2.0.

Usage

Since SQLGetTypeInfo() generates a result set and is equivalent to executing a query, it will generate a cursor and begin a transaction. To prepare and execute another statement on this statement handle, the cursor must be closed.

If SQLGetTypeInfo() is called with an invalid *DataType*, an empty result set is returned.

If either the **LONGDATACOMPAT** keyword or the SQL_ATTR_LONGDATA_COMPAT connection attribute is set, then SQL_LONGVARBINARY, SQL_LONGVARCHAR and SQL_LONGVARGRAPHIC will be returned for the *DATA_TYPE* argument instead of SQL_BLOB, SQL_CLOB and SQL_DBCLOB.

The columns of the result set generated by this function are described in the following section.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change. The data types returned are those that can be used in a CREATE TABLE, ALTER TABLE, DDL statement. Non-persistent data types such as the locator data types are not part of the returned result set. User-defined data types are not returned either.

Columns returned by SQLGetTypeInfo

Column 1 TYPE_NAME (VARCHAR(128) NOT NULL Data Type)
Data source-dependent data type name; for example, "CHAR()", "LONG VARBINARY". Applications must use this name in the CREATE TABLE and ALTER TABLE statements.

Column 2 DATA_TYPE (SMALLINT NOT NULL Data Type)
SQL data type define values, for example, SQL_VARCHAR, SQL_BLOB, SQL_DATE, SQL_INTEGER.

Column 3 COLUMN_SIZE (INTEGER Data Type)
If the data type is a character or binary string, then this column contains the maximum length in bytes; if it is a graphic (DBCS) string, this is the number of double byte characters for the column (the CLI/ODBC configuration keyword Graphic can change this default behaviour). If the data type is XML, zero is returned.

For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character.

For numeric data types, this is the total number of digits (precision).

Column 4 LITERAL_PREFIX (VARCHAR(128) Data Type)
Character that DB2 recognizes as a prefix for a literal of this data type. This column is null for data types where a literal prefix is not applicable.

Column 5 LITERAL_SUFFIX (VARCHAR(128) Data Type)
Character that DB2 recognizes as a suffix for a literal of this data type. This column is null for data types where a literal prefix is not applicable.

Column 6 CREATE_PARAMS (VARCHAR(128) Data Type)
The text of this column contains a list of keywords, separated by commas, corresponding to each parameter the application might specify in parenthesis when using the name in the TYPE_NAME column as a data

type in SQL. The keywords in the list can be LENGTH, PRECISION, or SCALE. They appear in the order that the SQL syntax requires that they be used.

A NULL indicator is returned if there are no parameters for the data type definition, (such as INTEGER).

Note: The intent of CREATE_PARAMS is to enable an application to customize the interface for a *DDL builder*. An application should expect, using this, only to be able to determine the number of arguments required to define the data type and to have localized text that could be used to label an edit control.

Column 7 NULLABLE (SMALLINT NOT NULL Data Type)
Indicates whether the data type accepts a NULL value
- Set to SQL_NO_NULLS if NULL values are disallowed.
- Set to SQL_NULLABLE if NULL values are allowed.
- Set to SQL_NULLABLE_UNKNOWN if it is not known whether NULL values are allowed or not.

Column 8 CASE_SENSITIVE (SMALLINT NOT NULL Data Type)
Indicates whether a character data type is case-sensitive in collations and comparisons. Valid values are SQL_TRUE and SQL_FALSE.

Column 9 SEARCHABLE (SMALLINT NOT NULL Data Type)
Indicates how the data type is used in a WHERE clause. Valid values are:
- SQL_UNSEARCHABLE : if the data type cannot be used in a WHERE clause.
- SQL_LIKE_ONLY : if the data type can be used in a WHERE clause only with the LIKE predicate.
- SQL_ALL_EXCEPT_LIKE : if the data type can be used in a WHERE clause with all comparison operators except LIKE.
- SQL_SEARCHABLE : if the data type can be used in a WHERE clause with any comparison operator.

Column 10 UNSIGNED_ATTRIBUTE (SMALLINT Data Type)
Indicates whether the data type is unsigned. The valid values are: SQL_TRUE, SQL_FALSE or NULL. A NULL indicator is returned if this attribute is not applicable to the data type.

Column 11 FIXED_PREC_SCALE (SMALLINT NOT NULL Data Type)
Contains the value SQL_TRUE if the data type is exact numeric and always has the same precision and scale; otherwise, it contains SQL_FALSE.

Column 12 AUTO_INCREMENT (SMALLINT Data Type)
Contains SQL_TRUE if a column of this data type is automatically set to a unique value when a row is inserted; otherwise, contains SQL_FALSE.

Column 13 LOCAL_TYPE_NAME (VARCHAR(128) Data Type)
This column contains any localized (native language) name for the data type that is different from the regular name of the data type. If there is no localized name, this column is NULL.

This column is intended for display only. The character set of the string is locale-dependent and is typically the default character set of the database.

Column 14 MINIMUM_SCALE (INTEGER Data Type)
The minimum scale of the SQL data type. If a data type has a fixed scale, the MINIMUM_SCALE and MAXIMUM_SCALE columns both contain the same value. NULL is returned where scale is not applicable.

Column 15 MAXIMUM_SCALE (INTEGER Data Type)
The maximum scale of the SQL data type. NULL is returned where scale is not applicable. If the maximum scale is not defined separately in the DBMS, but is defined instead to be the same as the maximum length of the column, then this column contains the same value as the COLUMN_SIZE column.

Column 16 SQL_DATA_TYPE (SMALLINT NOT NULL Data Type)
The value of the SQL data type as it appears in the SQL_DESC_TYPE field of the descriptor. This column is the same as the DATA_TYPE column (except for interval and datetime data types which CLI does not support).

Column 17 SQL_DATETIME_SUB (SMALLINT Data Type)
This field is always NULL (CLI does not support interval and datetime data types).

Column 18 NUM_PREC_RADIX (INTEGER Data Type)
If the data type is an approximate numeric type, this column contains the value 2 to indicate that COLUMN_SIZE specifies a number of bits. For exact numeric types, this column contains the value 10 to indicate that COLUMN_SIZE specifies a number of decimal digits. Otherwise, this column is NULL.

Column 19 INTERVAL_PRECISION (SMALLINT Data Type)
This field is always NULL (CLI does not support interval data types).

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 104. SQLGetTypeInfo SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle. *StatementHandle* had not been closed.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY004	SQL data type out of range.	An invalid *DataType* was specified.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Example

```
/* get data type information */
cliRC = SQLGetTypeInfo(hstmt, SQL_ALL_TYPES);
```

SQLMoreResults function (CLI) - Determine if there are more result sets

Determines whether there is more information available on the statement handle which has been associated with: an array input of parameter values for a query; a stored procedure that is returning result sets; or, batched SQL.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

Syntax

```
SQLRETURN   SQLMoreResults   (SQLHSTMT     StatementHandle);  /* hstmt */
```

Function arguments

Table 105. SQLMoreResults arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.

Usage

This function is used to return multiple results set in a sequential manner upon the execution of:

- a parameterized query with an array of input parameter values specified with the SQL_ATTR_PARAMSET_SIZE statement attribute and SQLBindParameter(), or
- a stored procedure containing SQL queries, the cursors of which have been left open so that the result sets remain accessible when the stored procedure has finished execution. For this scenario, the stored procedure is typically trying to return multiple result sets.
- or batched SQL. When multiple SQL statements are batched together during a single `SQLExecute()` or `SQLExecDirect()`.

After completely processing the first result set, the application can call SQLMoreResults() to determine if another result set is available. If the current result set has unfetched rows, SQLMoreResults() discards them by closing the cursor and, if another result set is available, returns SQL_SUCCESS.

If all the result sets have been processed, SQLMoreResults() returns SQL_NO_DATA_FOUND.

Applications that want to be able to manipulate more than one result set at the same time can use the CLI function `SQLNextResult()` to move a result set to another statement handle. `SQLNextResult()` does not support batched statements.

When using batched SQL, `SQLExecute()` or `SQLExecDirect()` will only execute the first SQL statement in the batch. `SQLMoreResults()` can then be called to execute the next SQL statement and will return SQL_SUCCESS if the next statement is

successfully executed. If there are no more statements to be executed, then SQL_NO_DATA_FOUND is returned. If the batched SQL statement is an UPDATE, INSERT, or DELETE statement, then `SQLRowCount()` can be called to determine the number of rows affected.

If `SQLCloseCursor()` or if SQLFreeStmt() is called with the SQL_CLOSE option, or SQLFreeHandle() is called with *HandleType* set to SQL_HANDLE_STMT, all pending result sets on this statement handle are discarded.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

Diagnostics

Table 106. SQLMoreResults SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

In addition SQLMoreResults() can return the SQLSTATEs associated with SQLExecute().

Example

```
cliRC = SQLMoreResults(hstmt);
```

SQLNativeSql function (CLI) - Get native SQL text

Shows how CLI interprets vendor escape clauses.

If the original SQL string passed in by the application contained vendor escape clause sequences, then CLI will return the transformed SQL string that would be seen by the data source (with vendor escape clauses either converted or discarded, as appropriate).

Specification:

- **CLI** 2.1
- **ODBC** 1.0

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLNativeSqlW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLNativeSql     (
               SQLHDBC           ConnectionHandle,  /* hdbc */
               SQLCHAR           *InStatementText,  /* szSqlStrIn */
               SQLINTEGER        TextLength1,       /* cbSqlStrIn */
               SQLCHAR           *OutStatementText, /* szSqlStr */
               SQLINTEGER        BufferLength,      /* cbSqlStrMax */
               SQLINTEGER        *TextLength2Ptr);  /* pcbSqlStr */
```

Function arguments

Table 107. SQLNativeSql arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection Handle
SQLCHAR *	*InStatementText*	input	Input SQL string
SQLINTEGER	*TextLength1*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store *InStatementText*.
SQLCHAR *	*OutStatementText*	output	Pointer to buffer for the transformed output string
SQLINTEGER	*BufferLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store *OutStatementText*.
SQLINTEGER *	*TextLength2Ptr*	output	The total number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function), excluding the null-terminator, available to return in *OutStatementText*. If the number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) available to return is greater than or equal to *BufferLength*, the output SQL string in *OutStatementText* is truncated to *BufferLength* - 1 SQLCHAR or SQLWCHAR elements.

Usage

This function is called when the application wishes to examine or display the transformed SQL string that would be passed to the data source by CLI. Translation (mapping) would only occur if the input SQL statement string contains vendor escape clause sequences.

CLI can only detect vendor escape clause syntax errors when SQLNativeSql() is called. Because CLI does not pass the transformed SQL string to the data source for preparation, syntax errors that are detected by the DBMS are not generated at this time. (The statement is not passed to the data source for preparation because the preparation may potentially cause the initiation of a transaction.)

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 108. SQLNativeSql SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer *OutStatementText* was not large enough to contain the entire SQL string, so truncation occurred. The argument *TextLength2Ptr* contains the total length of the untruncated SQL string. (Function returns with SQL_SUCCESS_WITH_INFO)
08003	Connection is closed.	The *ConnectionHandle* does not reference an open database connection.
37000	Invalid SQL syntax.	The input SQL string in *InStatementText* contained a syntax error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY009	Invalid argument value.	The argument *InStatementText* is a NULL pointer. The argument *OutStatementText* is a NULL pointer.
HY090	Invalid string or buffer length.	The argument *TextLength1* was less than 0, but not equal to SQL_NTS. The argument *BufferLength* was less than 0.

Restrictions

None.

SQLNumParams function (CLI) - Get number of parameters in a SQL statement

Returns the number of parameter markers in an SQL statement.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

Syntax

```
SQLRETURN   SQLNumParams (
               SQLHSTMT         StatementHandle,      /* hstmt */
               SQLSMALLINT      *ParameterCountPtr);  /* pcpar */
```

Function arguments

Table 109. SQLNumParams arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	Statement handle.
SQLSMALLINT *	*ParameterCountPtr*	Output	Number of parameters in the statement.

Usage

If the prepared SQL statement associated with *Statement Handle* contains batch SQL (multiple SQL statements separated by a semicolon ';'), the parameters are counted for the entire string and are not differentiated by the individual statements making up the batch.

This function can only be called after the statement associated with *StatementHandle* has been prepared. If the statement does not contain any parameter markers, *ParameterCountPtr* is set to 0.

An application can call this function to determine how many `SQLBindParameter()` (or `SQLBindFileToParam()`) calls are necessary for the SQL statement associated with the statement handle.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 110. SQLNumParams SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	This function was called before `SQLPrepare()` was called for the specified *StatementHandle* The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.

Table 110. SQLNumParams SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

SQLNextResult function (CLI) - Associate next result set with another statement handle

`SQLNextResult()` allows non-sequential access to multiple result sets returned from a stored procedure.

Specification:

- **CLI** 7.x

Syntax

```
SQLRETURN   SQLNextResult (SQLHSTMT StatementHandle1
                           SQLHSTMT StatementHandle2);
```

Function arguments

Table 111. SQLNextResult arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle1*	input	Statement handle.
SQLHSTMT	*StatementHandle2*	input	Statement handle.

Usage

A stored procedure returns multiple result sets by leaving one or more cursors open after exiting. The first result set is always accessed by using the statement handle that called the stored procedure. If multiple result sets are returned, either SQLMoreResults() or SQLNextResult() can be used to describe and fetch the result set.

SQLMoreResults() is used to close the cursor for the first result set and allow the next result set to be processed on the same statement handle, whereas SQLNextResult() moves the next result set to *StatementHandle2*, without closing the cursor on *StatementHandle1*. Both functions return SQL_NO_DATA_FOUND if there are no result sets to be fetched.

Using SQLNextResult() allows result sets to be processed in any order once they have been transferred to other statement handles. Mixed calls to SQLMoreResults() and SQLNextResult() are allowed until there are no more cursors (open result sets) on *StatementHandle1*.

When SQLNextResult() returns SQL_SUCCESS, the next result set is no longer associated with *StatementHandle1*. Instead, the next result set is associated with *StatementHandle2*, as if a call to SQLExecDirect() had just successfully executed a query on *StatementHandle2*. The cursor, therefore, can be described using SQLNumResultCols(), SQLDescribeCol(), or SQLColAttribute().

After SQLNextResult() has been called, the result set now associated with *StatementHandle2* is removed from the chain of remaining result sets and cannot be used again in either SQLNextResult() or SQLMoreResults(). This means that for 'n' result sets, SQLNextResult() can be called successfully at most 'n-1' times.

If `SQLCloseCursor()` or if SQLFreeStmt() is called with the SQL_CLOSE option, or SQLFreeHandle() is called with *HandleType* set to SQL_HANDLE_STMT, all pending result sets on this statement handle are discarded.

SQLNextResult() returns SQL_ERROR if *StatementHandle2* has an open cursor or *StatementHandle1* and *StatementHandle2* are not on the same connection. If any errors or warnings are returned, `SQLGetDiagRec()` must always be called on *StatementHandle1*.

Note: SQLMoreResults() also works with a parameterized query with an array of input parameter values specified with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute and SQLBindParameter(). SQLNextResult(), however, does not support this.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

Diagnostics

Table 112. SQLNextResult SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication Link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate the memory required to support execution or completion of the function.
HY010	Function sequence error.	The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. *StatementHandle2* has an open cursor associated with it. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access the memory required to support execution or completion of the function.

Table 112. SQLNextResult SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HYT00	Time-out expired.	The time-out period expired before the data source returned the result set. The time-out period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for SQLSetStmtAttr().

Restrictions

Only `SQLMoreResults()` can be used for parameterized queries and batched SQL.

Example

```
/* use SQLNextResult to push Result Set 2 onto the second statement handle */
cliRC = SQLNextResult(hstmt, hstmt2); /* open second cursor */
```

SQLNumResultCols function (CLI - Get number of result columns

Returns the number of columns in the result set associated with the input statement handle.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

SQLPrepare() or SQLExecDirect() must be called before calling this function.

After calling this function, you can call SQLColAttribute(), or one of the bind column functions.

Syntax

```
SQLRETURN   SQLNumResultCols (
               SQLHSTMT          StatementHandle,   /* hstmt */
               SQLSMALLINT       *ColumnCountPtr);  /* pccol */
```

Function arguments

Table 113. SQLNumResultCols arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLSMALLINT *	*ColumnCountPtr*	output	Number of columns in the result set

Usage

The function sets the output argument to zero if the last statement or function executed on the input statement handle did not generate a result set.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 114. SQLNumResultCols SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The function was called before calling SQLPrepare() or SQLExecDirect() for the *StatementHandle*. The function was called while in a data-at-execute (SQLParamData(), SQLPutData()) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Authorization

None.

Example

```
/* identify the number of output columns */
cliRC = SQLNumResultCols(hstmt, &nResultCols);
```

SQLParamData function (CLI) - Get next parameter for which a data value is needed

Sends long data in pieces, in conjunction with `SQLPutData()`. It can also be used to send fixed-length data at execution time.

Specification:

- **CLI** 2.1
- **ODBC** 1.0
- **ISO CLI**

Syntax

```
SQLRETURN   SQLParamData (
               SQLHSTMT         StatementHandle, /* hstmt */
               SQLPOINTER       *ValuePtrPtr );  /* prgbValue */
```

Function arguments

Table 115. SQLParamData arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLPOINTER *	*ValuePtrPtr*	output	Pointer to a buffer in which to return the address of the *ParameterValuePtr* buffer specified in `SQLBindParameter()` (for parameter data) or the address of the *TargetValuePtr* buffer specified in `SQLBindCol()` (for column data), as contained in the SQL_DESC_DATA_PTR descriptor record field.
		input	Starting in Version 9.7 Fix Pack 1, when SQL_ATTR_INTERLEAVED_PUTDATA is set to TRUE, this is an input argument. The application provides a value for which it wants to put data in subsequent `SQLPutData()` calls.

Usage

SQLParamData() returns SQL_NEED_DATA if there is at least one SQL_DATA_AT_EXEC parameter for which data still has not been assigned. This function returns an application-provided value in *ValuePtrPtr* supplied by the application during a previous `SQLBindParameter()` call. `SQLPutData()` is called one or more times (in the case of long data) to send the parameter data. SQLParamData() is called to signal that all the data has been sent for the current parameter and to advance to the next SQL_DATA_AT_EXEC parameter. SQL_SUCCESS is returned when all the parameters have been assigned data values and the associated statement has been executed successfully. If any errors occur during or before actual statement execution, SQL_ERROR is returned.

If SQLParamData() returns SQL_NEED_DATA, then only `SQLPutData()` or `SQLCancel()` calls can be made. All other function calls using this statement handle will fail. In addition, all function calls referencing the parent connection handle of *StatementHandle* will fail if they involve changing any attribute or state of that connection; that is, that following function calls on the parent connection handle are also not permitted:

- SQLSetConnectAttr()
- SQLEndTran()

However, calls to the SQLEndTran() function specifying SQL_ROLLBACK as completion type are allowed when the SQL_ATTR_FORCE_ROLLBACK connection attribute is set, the StreamPutData configuration keyword is set to 1, and autocommit mode is enabled.

Should they be invoked during an SQL_NEED_DATA sequence, these functions will return SQL_ERROR with SQLSTATE of **HY**010 and the processing of the SQL_DATA_AT_EXEC parameters will not be affected.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

- SQL_NEED_DATA

Diagnostics

SQLParamData() can return any SQLSTATE returned by the SQLPrepare(), SQLExecDirect(), and SQLExecute() functions. In addition, the following diagnostics can also be generated:

Table 116. SQLParamData SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	Transfer of data between CLI and the application variables would result in incompatible data conversion.
22026	String data, length mismatch	The SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo() was 'Y' and less data was sent for a long parameter (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or other long data type) than was specified with the *StrLen_or_IndPtr* argument in SQLBindParameter(). The SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo() was 'Y' and less data was sent for a long column (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or other long data type) than was specified in the length buffer corresponding to a column in a row of data that was updated with SQLSetPos().
40001	Transaction rollback.	The transaction to which this SQL statement belonged was rolled back due to a deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY000	General error.	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec() in the argument *MessageText* describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, SQLCancel() was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	SQLParamData() was called out of sequence. This call is only valid after an SQLExecDirect() or an SQLExecute(), or after an SQLPutData() call. Even though this function was called after an SQLExecDirect() or an SQLExecute() call, there were no SQL_DATA_AT_EXEC parameters (left) to process.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY092	Option type out of range.	The *FileOptions* argument of a previous SQLBindFileToParam() operation was not valid.
HY506	Error closing a file.	Error encountered while trying to close a temporary file.

Table 116. SQLParamData SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY509	Error deleting a file.	Error encountered while trying to delete a temporary file.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

Example

```
/* get next parameter for which a data value is needed */
cliRC = SQLParamData(hstmt, (SQLPOINTER *)&valuePtr);
```

SQLParamOptions function (CLI) - Specify an input array for a parameter

In ODBC 3.0, SQLParamOptions() has been deprecated and replaced with SQLSetStmtAttr().

Although this version of CLI continues to support SQLParamOptions(), use SQLSetStmtAttr() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

The statement:

```
SQLParamOptions(hstmt, crow, pirow);
```

for example, would be rewritten using the new function as:

```
SQLSetStmtAttr(hstmt, fOption, pvParam, fStrLen);
```

SQLPrepare function (CLI) - Prepare a statement

`SQLPrepare()` associates an SQL statement or XQuery expression with the input statement handle provided.

The application can include one or more parameter markers in the SQL statement. To include a parameter marker, the application embeds a question mark (?) or a colon followed by a name (*:name*) into the SQL string at the appropriate position. The application can reference this prepared statement by passing the statement handle to other functions.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

Note: For XQuery expressions, you cannot specify parameter markers in the expression itself. You can, however, use the XMLQUERY function to bind

parameter markers to XQuery variables. The values of the bound parameter markers will then be passed to the XQuery expression specified in XMLQUERY for execution.

If the statement handle has been previously used with a query statement (or any function that returns a result set), either `SQLCloseCursor()` or `SQLFreeStmt()` with the SQL_CLOSE option must be called to close the cursor before calling SQLPrepare().

XQuery expressions must be prefixed with the "XQUERY" keyword. To prepare and execute XQuery expressions without having to include this keyword, set the statement attribute SQL_ATTR_XQUERY_STATEMENT to SQL_TRUE before calling `SQLPrepare()` or `SQLExecDirect()`.

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLPrepareW(). See “Unicode functions (CLI)” on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLPrepare (
               SQLHSTMT          StatementHandle,  /* hstmt */
               SQLCHAR           *StatementText,   /* szSqlStr */
               SQLINTEGER        TextLength);      /* cbSqlStr */
```

Function arguments

Table 117. SQLPrepare arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle. There must not be an open cursor associated with *StatementHandle*.
SQLCHAR *	*StatementText*	input	SQL statement string
SQLINTEGER	*TextLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the *StatementText* argument, or SQL_NTS if *StatementText* is null-terminated.

Usage

Deferred prepare is on by default. The prepare request is not sent to the server until either `SQLDescribeParam()`, `SQLExecute()`, `SQLNumResultCols()`, `SQLDescribeCol()`, or `SQLColAttribute()` is called using the same statement handle as the prepared statement. This minimizes network flow and improves performance.

If the SQL statement text contains vendor escape clause sequences, CLI will first modify the SQL statement text to the appropriate DB2 specific format before submitting it to the database for preparation. If the application does not generate SQL statements that contain vendor escape clause sequences then the SQL_ATTR_NOSCAN statement attribute should be set to SQL_NOSCAN at the connection level so that CLI does not perform a scan for any vendor escape clauses.

Once a statement has been prepared using SQLPrepare(), the application can request information about the format of the result set (if the statement was a query) by calling:

- `SQLNumResultCols()`
- `SQLDescribeCol()`
- `SQLColAttribute()`

Information about the parameter markers in *StatementText* can be requested using the following functions:

- `SQLDescribeParam()`
- `SQLNumParams()`

Note: The first invocation of `SQLNumResultCols()`, `SQLDescribeCol()`, `SQLColAttribute()`, or `SQLDescribeParam()` will force the PREPARE request to be sent to the server if deferred prepare is enabled.

The SQL statement string might contain parameter markers and `SQLNumParams()` can be called to determine the number of parameter markers in the statement. A parameter marker is represented by a "?" character or a colon followed by a name (*:name*), and is used to indicate a position in the statement where an application-supplied value is to be substituted when `SQLExecute()` is called. The bind parameter functions, `SQLBindParameter()`, `SQLSetParam()` and `SQLBindFileToParam()`, are used to bind or associate application variables with each parameter marker and to indicate if any data conversion should be performed at the time the data is transferred. An application can call `SQLDescribeParam()` to retrieve information about the data expected by the database server for the parameter marker.

All parameters must be bound before calling `SQLExecute()`.

Refer to the PREPARE statement for information about rules related to parameter markers.

Once the application has processed the results from the `SQLExecute()` call, it can execute the statement again with new (or the same) parameter values.

The SQL statement can be COMMIT or ROLLBACK and executing either of these statements has the same effect as calling SQLEndTran() on the current connection handle.

If the SQL statement is a positioned DELETE or a positioned UPDATE, the cursor referenced by the statement must be defined on a separate statement handle under the same connection handle and same isolation level.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 118. SQLPrepare SQLSTATEs

SQLSTATE	Description	Explanation
01504	The UPDATE or DELETE statement does not include a WHERE clause.	*StatementText* contained an UPDATE or DELETE statement which did not contain a WHERE clause.
01508	Statement disqualified for blocking.	The statement was disqualified for blocking for reasons other than storage.
21S01	Insert value list does not match column list.	*StatementText* contained an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	*StatementText* contained a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
22018	Invalid character value for cast specification.	*StatementText* contained an SQL statement that contained a literal or parameter and the value was incompatible with the data type of the associated table column.
22019	Invalid escape character	The argument *StatementText* contained a LIKE predicate with an ESCAPE in the WHERE clause, and the length of the escape character following ESCAPE was not equal to 1.
22025	Invalid escape sequence	The argument *StatementText* contained "LIKE *pattern value* ESCAPE *escape character*" in the WHERE clause, and the character following the escape character in the pattern value was not one of "%" or "_".
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
34000	Invalid cursor name.	*StatementText* contained a positioned DELETE or a positioned UPDATE and the cursor referenced by the statement being executed was not open.
37xxx [a]	Invalid SQL syntax.	*StatementText* contained one or more of the following issues: • an SQL statement that the connected database server could not prepare • a statement containing a syntax error
40001	Transaction rollback.	The transaction to which this SQL statement belonged was rolled back due to deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
42xxx [a]	Syntax Error or Access Rule Violation.	**42**5xx indicates the authorization ID does not have permission to execute the SQL statement contained in *StatementText*. Other **42**xxx SQLSTATES indicate a variety of syntax or access problems with the statement.
58004	Unexpected system failure.	Unrecoverable system error.
S0001	Database object already exists.	*StatementText* contained a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already existed.
S0002	Database object does not exist.	*StatementText* contained an SQL statement that references a table name or a view name which did not exist.
S0011	Index already exists.	*StatementText* contained a CREATE INDEX statement and the specified index name already existed.
S0012	Index not found.	*StatementText* contained a DROP INDEX statement and the specified index name did not exist.

Table 118. SQLPrepare SQLSTATEs (continued)

SQLSTATE	Description	Explanation
S0021	Column already exists.	*StatementText* contained an ALTER TABLE statement and the column specified in the ADD clause was not unique or identified an existing column in the base table.
S0022	Column not found.	*StatementText* contained an SQL statement that references a column name which did not exist.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY009	Invalid argument value.	*StatementText* was a null pointer.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The argument *TextLength* was less than 1, but not equal to SQL_NTS.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Note:

a xxx refers to any SQLSTATE with that class code. Example, **37**xxx refers to any SQLSTATE in the **37** class.

Note: Not all DBMSs report all of the listed diagnostic messages at prepare time. If deferred prepare is left on as the default behavior (controlled by the SQL_ATTR_DEFERRED_PREPARE statement attribute), then these errors could occur when the PREPARE is flowed to the server. The application must be able to handle these conditions when calling functions that cause this flow. These functions include `SQLExecute()`, `SQLDescribeParam()`, `SQLNumResultCols()`, `SQLDescribeCol()`, and `SQLColAttribute()`.

Authorization

None.

Example

```
SQLCHAR *stmt = (SQLCHAR *)"DELETE FROM org WHERE deptnumb = ? ";

/* ... */
```

```
/* prepare the statement */
cliRC = SQLPrepare(hstmt, stmt, SQL_NTS);
```

SQLPrimaryKeys function (CLI) - Get primary key columns of a table

The SQLPrimaryKeys() function returns a list of column names that comprise the primary key for a table.

The information is returned in an SQL result set, which you can retrieve by using the same functions that you use to process a result set that is generated by a query.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

SQLPrimaryKeys() returns a list of column names that comprise the primary key for a table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is SQLPrimaryKeysW(). For information about ANSI to Unicode function mappings, see"Unicode functions (CLI)" on page 5.

Syntax

```
SQLRETURN   SQLPrimaryKeys   (
               SQLHSTMT          StatementHandle,   /* hstmt */
               SQLCHAR           *CatalogName,      /* szCatalogName */
               SQLSMALLINT       NameLength1,       /* cbCatalogName */
               SQLCHAR           *SchemaName,       /* szSchemaName */
               SQLSMALLINT       NameLength2,       /* cbSchemaName */
               SQLCHAR           *TableName,        /* szTableName */
               SQLSMALLINT       NameLength3);      /* cbTableName */
```

Function arguments

Table 119. SQLPrimaryKeys arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	The statement handle.
SQLCHAR *	*CatalogName*	Input	A catalog qualifier of a 3-part table name. If the target DBMS does not support 3-part naming, and *CatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS is returned. Otherwise, this is a valid filter for DBMSs that supports 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.
SQLCHAR *	*SchemaName*	Input	The schema qualifier of table name.
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*TableName*	Input	The table name.

Table 119. SQLPrimaryKeys arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *TableName*, or SQL_NTS if *TableName* is null-terminated.

Usage

The `SQLPrimaryKeys()` function returns the primary key columns from a single table. You cannot use search patterns to specify any of the arguments.

The result set contains the columns that are listed in Columns Returned By SQLPrimaryKeys, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME and ORDINAL_POSITION columns.

In many cases, calls to the `SQLPrimaryKeys()` function map to a complex and thus expensive query against the system catalog, so you should use these calls sparingly, and save the results rather than repeating calls.

If the schema name is not provided, the schema name defaults to the name that is in effect for the current connection.

Call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns that are supported by the connected DBMS.

You can specify *ALL as a value in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. CLI searches on all existing schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns Returned By SQLPrimaryKeys

Column 1 TABLE_CAT (VARCHAR(128))
The primary key table catalog name. The value is NULL if this table does not have catalogs.

Column 2 TABLE_SCHEM (VARCHAR(128))
The name of the schema that contains TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128) not NULL)
The name of the specified table.

Column 4 COLUMN_NAME (VARCHAR(128) not NULL)
The primary key column name.

Column 5 KEY_SEQ (SMALLINT not NULL)
The column sequence number in the primary key, starting with 1.

Column 6 PK_NAME (VARCHAR(128))
The primary key identifier. NULL if not applicable to the data source.

Note: The column names that are used by CLI follow the X/Open CLI CAE specification style. The column types, contents, and order are identical to those defined for the `SQLPrimaryKeys()` result set in ODBC.

If the specified table does not contain a primary key, an empty result set is returned.

Return codes

- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_STILL_EXECUTING
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

Diagnostics

Table 120. SQLPrimaryKeys SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while in a BEGIN COMPOUND and END COMPOUND SQL operation.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS.
HYC00	Driver not capable.	CLI does not support *catalog* as a qualifier for table name.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. You can set the timeout period by using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

Example

```
/* get the primary key columns of a table */
cliRC = SQLPrimaryKeys(hstmt, NULL, 0, tbSchema, SQL_NTS, tbName, SQL_NTS);
```

SQLProcedureColumns function (CLI) - Get input/output parameter information for a procedure

The SQLProcedureColumns() function returns a list of input and output parameters that are associated with a stored procedure.

The information is returned in an SQL result set, which you can retrieve using the same functions that you use to process a result set that is generated by a query.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is SQLProcedureColumnsW(). For information about ANSI to Unicode function mappings, see "Unicode functions (CLI)" on page 5.

Syntax

```
SQLRETURN SQLProcedureColumns(
            SQLHSTMT          StatementHandle,   /* hstmt */
            SQLCHAR           *CatalogName,      /* szProcCatalog */
            SQLSMALLINT       NameLength1,       /* cbProcCatalog */
            SQLCHAR           *SchemaName,       /* szProcSchema */
            SQLSMALLINT       NameLength2,       /* cbProcSchema */
            SQLCHAR           *ProcName,         /* szProcName */
            SQLSMALLINT       NameLength3,       /* cbProcName */
            SQLCHAR           *ColumnName,       /* szColumnName */
            SQLSMALLINT       NameLength4);      /* cbColumnName */
```

Function arguments

Table 121. SQLProcedureColumns arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	The statement handle.
SQLCHAR *	*CatalogName*	Input	A catalog qualifier of a 3-part table name. If the target DBMS does not support 3-part naming, and *CatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS is returned. Otherwise, this is a valid filter for DBMSs that supports 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.

Table 121. SQLProcedureColumns arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*SchemaName*	Input	A buffer that can contain a *pattern value* to qualify the result set by schema name. For DB2 for z/OS, the stored procedures are in one schema; the only acceptable value for the *SchemaName* argument is a null pointer. If a value is specified, an empty result set and SQL_SUCCESS are returned. For DB2 Database for Linux, UNIX, and Windows, *SchemaName* can contain a valid pattern value. For more information about valid search patterns, see the catalog functions input arguments.
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*ProcName*	Input	A buffer that can contain a *pattern value* to qualify the result set by the procedure name.
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *ProcName*, or SQL_NTS if *ProcName* is null-terminated.
SQLCHAR *	*ColumnName*	Input	A buffer that can contain a *pattern value* to qualify the result set by the parameter name. Use this argument to further qualify the result set that is already restricted by specifying a non-empty value for ProcName, SchemaName, or both.
SQLSMALLINT	*NameLength4*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *ColumnName*, or SQL_NTS if *ColumnName* is null-terminated.

Usage

The `SQLProcedureColumns()` function returns the information in a result set, ordered by PROCEDURE_CAT, PROCEDURE_SCHEM, PROCEDURE_NAME, and COLUMN_TYPE. Columns returned by SQLProcedureColumns lists the columns in the result set. Applications should be aware that columns that are beyond the last column might be defined in future releases.

In many cases, calls to the `SQLProcedureColumns()` function map to a complex and thus expensive query against the system catalog, so you should use the calls sparingly, and save the results rather than repeating calls.

Call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, and COLUMN_NAME columns that are supported by the connected DBMS.

If the SQL_ATTR_LONGDATA_COMPAT connection attribute is set, LOB column types are reported as LONG VARCHAR, LONG VARBINARY or LONG VARGRAPHIC types.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

If the stored procedure is at a DB2 for z/OS server, the name of the stored procedure must be registered in the SYSIBM.SYSPROCEDURES catalog table of the server. For V8 and later servers, the stored procedures must be registered in the SYSIBM.SYSROUTINES and SYSIBM.SYSPARAMS catalog tables of the server.

For versions of other DB2 servers that do not provide facilities for a stored procedure catalog, an empty result set is returned.

CLI returns information about the input, input/output, and output parameters that are associated with the stored procedure, but cannot return descriptor information for any result sets that the stored procedure might return.

You can specify *ALL as a value in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. CLI searches on all existing schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL.

Columns returned by SQLProcedureColumns

Column 1 PROCEDURE_CAT (VARCHAR(128))
The procedure catalog name. The value is NULL if this procedure does not have catalogs.

Column 2 PROCEDURE_SCHEM (VARCHAR(128))
The name of the schema that contains PROCEDURE_NAME. This is NULL for DB2 for z/OS `SQLProcedureColumns()` result sets.

Column 3 PROCEDURE_NAME (VARCHAR(128))
The name of the procedure.

Column 4 COLUMN_NAME (VARCHAR(128))
The name of the parameter.

Column 5 COLUMN_TYPE (SMALLINT not NULL)
Identifies the type of information that is associated with this row. The values that can be returned are:

- SQL_PARAM_INPUT is an input parameter.
- SQL_PARAM_INPUT_OUTPUT is an input / output parameter.
- SQL_PARAM_OUTPUT is an output parameter.

The values which are defined in the ODBC specification but are not returned:

- SQL_PARAM_TYPE_UNKNOWN : the parameter type is unknown.
- SQL_RETURN_VALUE is the return value of the procedure, in the procedure column.
- SQL_RESULT_COL is a column in the result set.

Column 6 DATA_TYPE (SMALLINT not NULL)
The SQL data type.

Column 7 TYPE_NAME (VARCHAR(128) not NULL)
The character string that represents the name of the data type that corresponds to DATA_TYPE.

Column 8 COLUMN_SIZE (INTEGER)

For XML arguments in SQL routines, zero is returned (as XML arguments have no length). For cataloged external routines, however, XML parameters are declared as XML AS CLOB(n), in which case COLUMN_SIZE is the cataloged length, n.

If the DATA_TYPE column value denotes a character or binary string, this column contains the maximum length in SQLCHAR or SQLWCHAR elements. If DATA_TYPE column value is a graphic (DBCS) string, the COLUMN_SIZE is the number of double byte SQLCHAR or SQLWCHAR elements for the parameter.

For date, time, and timestamp data types, this is the total number of SQLCHAR or SQLWCHAR elements that are required to display the value when converted to character data type.

For numeric data types, COLUMN_SIZE value is either the total number of digits or the total number of bits that are allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.

See the table of data type precision.

Column 9 BUFFER_LENGTH (INTEGER)

The maximum number of bytes for the associated C buffer to store data from this parameter if SQL_C_DEFAULT is specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` calls. This length excludes any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign.

For XML arguments in SQL routines, zero is returned (as XML arguments have no length). For cataloged external routines, however, XML parameters are declared as XML AS CLOB(n), in which case BUFFER_LENGTH is the cataloged length, n.

See the table of data type length.

Column 10 DECIMAL_DIGITS (SMALLINT)

The scale of the parameter. NULL is returned for data types where scale is not applicable.

See the table of data type scale.

Column 11 NUM_PREC_RADIX (SMALLINT)

Either 10, 2, or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, and the COLUMN_SIZE column contains the number of bits that are allowed in the parameter.

If DATA_TYPE is an exact numeric data type, this column contains the value 10, and the COLUMN_SIZE and DECIMAL_DIGITS columns contain the number of decimal digits that are allowed for the parameter.

For numeric data types, the DBMS can return a NUM_PREC_RADIX of either 10 or 2.

NULL is returned for data types where radix is not applicable.

Column 12 NULLABLE (SMALLINT not NULL)

SQL_NO_NULLS if the parameter does not accept NULL values.

SQL_NULLABLE if the parameter accepts NULL values.

Column 13 REMARKS (VARCHAR(254))

Might contain descriptive information about the parameter.

Column 14 COLUMN_DEF (VARCHAR)
The default value of the column.

If NULL was specified as the default value, this column is the word NULL, not enclosed in quotation marks. If the default value cannot be represented without truncation, this column contains TRUNCATED, not enclosed in single quotation marks. If no default value is specified, this column is NULL.

You can use the value of COLUMN_DEF to generate a new column definition, except when COLUMN_DEF contains the value TRUNCATED.

Column 15 SQL_DATA_TYPE (SMALLINT not NULL)
The value of the SQL data type as it is displayed in the SQL_DESC_TYPE field of the descriptor. This column is the same as the DATA_TYPE column except for datetime data types (CLI does not support interval data types).

For datetime data types, the SQL_DATA_TYPE field in the result set is SQL_DATETIME, and the SQL_DATETIME_SUB field returns the subcode for the specific datetime data type (SQL_CODE_DATE, SQL_CODE_TIME or SQL_CODE_TIMESTAMP).

Column 16 SQL_DATETIME_SUB (SMALLINT)
The subtype code for datetime data types. For all other data types this column returns a NULL value (including interval data types that CLI does not support).

Column 17 CHAR_OCTET_LENGTH (INTEGER)
The maximum length in bytes of a character data type column. For all other data types, this column returns a NULL.

Column 18 ORDINAL_POSITION (INTEGER NOT NULL)
Contains the ordinal position of the parameter that is given by COLUMN_NAME in this result set. This is the ordinal position of the argument to be provided on the CALL statement. The leftmost argument has an ordinal position of 1.

Column 19 IS_NULLABLE (Varchar)
- "NO" if the column cannot contain NULLs.
- "YES" if the column can include NULLs.
- Zero-length string if nullability is unknown.

ISO rules are followed to determine nullability.

An ISO SQL-compliant DBMS cannot return an empty string.

The value that is returned for this column is different than the value that is returned for the NULLABLE column. (See the description of the NULLABLE column.)

Note:

- The column names that are used by CLI follow the X/Open CLI CAE specification style. The column types, contents, and order are identical to those defined for the `SQLProcedureColumns()` result set in ODBC.
- If two modules contain procedures that share the same name, `SQLProcedureColumns()` returns details about both procedures.

Return codes

- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_STILL_EXECUTING

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

Diagnostics

Table 122. SQLProcedureColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
42601	PARMLIST syntax error.	The PARMLIST value in the stored procedures catalog table contains a syntax error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while in a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. You can set the timeout period by using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

The `SQLProcedureColumns()` function does not return information about the attributes of result sets that might be returned from stored procedures.

If an application is connected to a DB2 server that does not provide support for a stored procedure catalog, or does not provide support for stored procedures, the `SQLProcedureColumns()` function returns an empty result set.

Example

```
/* get input/output parameter information for a procedure */
sqlrc = SQLProcedureColumns(hstmt,
                            NULL,
                            0, /* catalog name not used */
                            (unsigned char *)colSchemaNamePattern,
                            SQL_NTS, /* schema name not currently used */
                            (unsigned char *)procname,
                            SQL_NTS,
                            colNamePattern,
                            SQL_NTS); /* all columns */
```

SQLProcedures function (CLI) - Get list of procedure names

The SQLProcedures() function returns a list of stored procedure names that have been registered at the server, and which match the specified search pattern.

The information is returned in an SQL result set, which you can retrieve by using the same functions that you use to process a result set that is generated by a query.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is SQLProceduresW(). For information about ANSI to Unicode function mappings, see "Unicode functions (CLI)" on page 5.

Syntax

```
SQLRETURN   SQLProcedures    (
               SQLHSTMT          StatementHandle,   /* hstmt */
               SQLCHAR           *CatalogName,      /* szProcCatalog */
               SQLSMALLINT       NameLength1,       /* cbProcCatalog */
               SQLCHAR           *SchemaName,       /* szProcSchema */
               SQLSMALLINT       NameLength2,       /* cbProcSchema */
               SQLCHAR           *ProcName,         /* szProcName */
               SQLSMALLINT       NameLength3);      /* cbProcName */
```

Function arguments

Table 123. SQLProcedures arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	The statement handle.
SQLCHAR *	*CatalogName*	Input	A catalog qualifier of a 3-part table name. If the target DBMS does not support 3-part naming, and *CatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS is returned. Otherwise, this is a valid filter for DBMSs that support 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.

Table 123. SQLProcedures arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*SchemaName*	Input	A buffer that can contain a *pattern value* to qualify the result set by schema name. For DB2 for z/OS, the stored procedures are in one schema; the only acceptable value for the *SchemaName* argument is a null pointer. If a value is specified, an empty result set and SQL_SUCCESS are returned. For DB2 Database for Linux, UNIX, and Windows, *SchemaName* can contain a valid pattern value. For more information about valid search patterns, see the catalog functions input arguments.
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*ProcName*	Input	A buffer that can contain a *pattern value* to qualify the result set by table name.
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *ProcName*, or SQL_NTS if *ProcName* is null-terminated.

Usage

The result set that is returned by the `SQLProcedures()` function contains the columns that are listed in Columns returned by SQLProcedures in the order given. The rows are ordered by PROCEDURE_CAT, PROCEDURE_SCHEMA, and PROCEDURE_NAME.

In many cases, calls to the `SQLProcedures()` function map to a complex and thus expensive query against the system catalog, so you should use them sparingly, and save the results rather than repeating calls.

Call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns that are supported by the connected DBMS.

If the SQL_ATTR_LONGDATA_COMPAT connection attribute is set, LOB column types are reported as LONG VARCHAR, LONG VARBINARY, or LONG VARGRAPHIC types.

If the stored procedure is at a DB2 for z/OS server, the name of the stored procedures must be registered in the server's SYSIBM.SYSPROCEDURES catalog table. For V8 and later servers, the stored procedure must be registered in the server's SYSIBM.SYSROUTINES and SYSIBM.SYSPARAMS catalog tables.

For other versions of DB2 servers that do not provide facilities for a stored procedure catalog, an empty result set is returned.

You can specify *ALL as a value in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. CLI searches on all existing

schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns returned by SQLProcedures

Column 1 PROCEDURE_CAT (VARCHAR(128))
The procedure catalog name. The value is NULL if this procedure does not have catalogs.

Column 2 PROCEDURE_SCHEM (VARCHAR(128))
The name of the schema that contains PROCEDURE_NAME.

Column 3 PROCEDURE_NAME (VARCHAR(128) NOT NULL)
The name of the procedure.

Column 4 NUM_INPUT_PARAMS (INTEGER not NULL)
The number of input parameters. INOUT parameters are not counted as part of this number.

To determine information regarding INOUT parameters, examine the COLUMN_TYPE column that is returned by `SQLProcedureColumns()`.

Column 5 NUM_OUTPUT_PARAMS (INTEGER not NULL)
The number of output parameters. INOUT parameters are not counted as part of this number.

To determine information regarding INOUT parameters, examine the COLUMN_TYPE column that is returned by `SQLProcedureColumns()`.

Column 6 NUM_RESULT_SETS (INTEGER not NULL)
The number of result sets that are returned by the procedure.

You should not use this column, it is reserved for future use by ODBC.

Column 7 REMARKS (VARCHAR(254))
Contains the descriptive information about the procedure.

Column 8 PROCEDURE_TYPE (SMALLINT)
Defines the procedure type:
- SQL_PT_UNKNOWN: It cannot be determined whether the procedure returns a value.
- SQL_PT_PROCEDURE: The returned object is a procedure that does not have a return value.
- SQL_PT_FUNCTION: The returned objects is a function that has a return value

CLI always returns SQL_PT_PROCEDURE.

Note:

- The column names that are used by CLI follow the X/Open CLI CAE specification style. The column types, contents, and order are identical to those that are defined for the `SQLProcedures()` result set in ODBC.
- If two modules contain procedures that share the same name, the `SQLProcedures()` function returns details about both procedures.

Return codes

- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_STILL_EXECUTING
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

Diagnostics

Table 124. SQLProcedures SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. The function was called again on *StatementHandle*.
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. You can set timeout period by using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

If an application is connected to a DB2 server that does not provide support for a stored procedure catalog, or does not provide support for stored procedures, `SQLProcedureColumns()` will return an empty result set.

SQLPutData function (CLI) - Passing data value for a parameter

Sends large parameter values in pieces. SQLPutData() is called following an SQLParamData() call returning SQL_NEED_DATA to supply parameter data values.

Specification:

- **CLI** 2.1
- **ODBC** 1.0
- **ISO CLI**

Syntax

```
SQLRETURN   SQLPutData       (
               SQLHSTMT          StatementHandle,   /* hstmt */
               SQLPOINTER        DataPtr,           /* rgbValue */
               SQLLEN            StrLen_or_Ind);    /* cbValue */
```

Function arguments

Table 125. SQLPutData arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	Statement handle.
SQLPOINTER	*DataPtr*	Input	Pointer to the actual data, or portion of data, for a parameter. The data must be in the form specified in the SQLBindParameter() call that the application used when specifying the parameter.
SQLLEN	*StrLen_or_Ind*	Input	The length of *DataPtr*. Specifies the amount of data sent in a call to SQLPutData() . The amount of data can vary with each call for a given parameter. The application can also specify SQL_NTS or SQL_NULL_DATA for *StrLen_or_Ind*. *StrLen_or_Ind* is ignored for all fixed length C buffer types, such as date, time, timestamp, and all numeric C buffer types. For cases where the C buffer type is SQL_C_CHAR or SQL_C_BINARY, or if SQL_C_DEFAULT is specified as the C buffer type and the C buffer type default is SQL_C_CHAR or SQL_C_BINARY, this is the number of bytes of data in the *DataPtr* buffer.

Usage

The application calls SQLPutData() after calling SQLParamData() on a statement in the SQL_NEED_DATA state to supply the data values for an SQL_DATA_AT_EXEC parameter. Long data can be sent in pieces via repeated calls to SQLPutData(). CLI generates a temporary file for each SQL_DATA_AT_EXEC parameter to which each piece of data is appended when SQLPutData() is called. The path in which CLI creates its temporary files can be set using the TEMPDIR keyword in the `db2cli.ini` file. If this keyword is not set, CLI attempts to write to the path specified by the environment variables TEMP or TMP. After all the pieces of data for the parameter have been sent, the application calls SQLParamData() again to proceed to the next SQL_DATA_AT_EXEC parameter, or, if all parameters have data values, to execute the statement.

SQLPutData() cannot be called more than once for a fixed length C buffer type, such as SQL_C_LONG.

After an SQLPutData() call, the only legal function calls are SQLParamData(), SQLCancel(), or another SQLPutData() if the input data is character or binary data. As with SQLParamData(), all other function calls using this statement handle will fail. In addition, all function calls referencing the parent connection handle of *StatementHandle* will fail if they involve changing any attribute or state of that connection; that is, the following function calls on the parent connection handle are also not permitted:
- SQLSetConnectAttr()
- SQLEndTran()

However, calls to the SQLEndTran() function specifying SQL_ROLLBACK as completion type are allowed when the SQL_ATTR_FORCE_ROLLBACK connection attribute is set, the StreamPutData configuration keyword is set to 1, and autocommit mode is enabled.

Should they be invoked during an SQL_NEED_DATA sequence, these functions will return SQL_ERROR with SQLSTATE of **HY**010 and the processing of the SQL_DATA_AT_EXEC parameters will not be affected.

If one or more calls to SQLPutData() for a single parameter results in SQL_SUCCESS, attempting to call SQLPutData() with *StrLen_or_Ind* set to SQL_NULL_DATA for the same parameter results in an error with SQLSTATE of **22**005. This error does not result in a change of state; the statement handle is still in a *Need Data* state and the application can continue sending parameter data.

Return codes
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Some of the following diagnostics conditions might also be reported on the final SQLParamData() call rather than at the time the SQLPutData() is called.

Table 126. SQLPutData SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data sent for a numeric parameter was truncated without the loss of significant digits. Timestamp data sent for a date or time column was truncated. Function returns with SQL_SUCCESS_WITH_INFO.
22001	String data right truncation.	More data was sent for a binary or char data than the data source can support for that column.
22003	Numeric value out of range.	The data sent for a numeric parameter caused the whole part of the number to be truncated when assigned to the associated column. SQLPutData() was called more than once for a fixed length parameter.

Table 126. SQLPutData SQLSTATEs (continued)

SQLSTATE	Description	Explanation
22005	Error in assignment.	The data sent for a parameter was incompatible with the data type of the associated table column.
22007	Invalid datetime format.	The data value sent for a date, time, or timestamp parameters was invalid.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY009	Invalid argument value.	The argument *DataPtr* was a NULL pointer, and the argument *StrLen_or_Ind* was neither 0 nor SQL_NULL_DATA.
HY010	Function sequence error.	The statement handle *StatementHandle* must be in a need data state and must have been positioned on an SQL_DATA_AT_EXEC parameter via a previous SQLParamData() call.
HY090	Invalid string or buffer length.	The argument *DataPtr* was not a NULL pointer, and the argument *StrLen_or_Ind* was less than 0, but not equal to SQL_NTS or SQL_NULL_DATA.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

A additional value for *StrLen_or_Ind*, SQL_DEFAULT_PARAM, was introduced in ODBC 2.0, to indicate that the procedure is to use the default value of a parameter, rather than a value sent from the application. Since DB2 stored procedure arguments do not support default values, specification of this value for StrLen_or_Ind argument will result in an error when the CALL statement is executed since the SQL_DEFAULT_PARAM value will be considered an invalid length.

ODBC 2.0 also introduced the SQL_LEN_DATA_AT_EXEC(*length*) macro to be used with the *StrLen_or_Ind argument*. The macro is used to specify the sum total length of the entire data that would be sent for character or binary C data via the subsequent SQLPutData() calls. Since the DB2 ODBC driver does not need this information, the macro is not needed. An ODBC application calls SQLGetInfo() with the SQL_NEED_LONG_DATA_LEN option to check if the driver needs this information. The DB2 ODBC driver will return 'N' to indicate that this information is not needed by SQLPutData().

Example

```
SQLCHAR buffer[BUFSIZ];
size_t n = BUFSIZ;
```

```
/* ... */

/* passing data value for a parameter */
cliRC = SQLPutData(hstmt, buffer, n);
```

SQLReloadConfig function (CLI) - Reload a configuration property from the client configuration file

The SQLReloadConfig() function reloads a configuration property from the db2dsdriver.cfg client configuration file.

Specification:

- **CLI** 9.7

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is SQLReloadConfigW(). For more information about ANSI to Unicode function mappings, see "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLReloadConfig(SQLINTEGER       ConfigProperty,
                            SQLCHAR          *DiagInfoString,
                            SQLSMALLINT      BufferLength,
                            SQLSMALLINT      *StringLengthPtr);
```

Function arguments

Table 127. SQLReloadConfig function arguments

Data type	Argument	Use	Description
SQLINTEGER	*ConfigProperty*	Input	A predefined grouping of db2dsdriver.cfg file sections to reload. **DSD_ACR_AFFINITY** is the only supported value. **DSD_ACR_AFFINITY** value is defined in the usage section.
SQLCHAR *	*DiagInfoString*	Output	If the value is SQL_ERROR, CLI returns a detailed error description or information in the **DiagInfoString** output. If the value is SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, CLI does not return any information. In DB2 Version 9.7 Fix Pack 5 and later fix packs, warning messages are prefixed with a diagnostic string that consists of the database name, server name, and port number (*database:hostname:port*). Multiple warnings are separated with newline characters.
SQLINTEGER	*BufferLength*	Input	The length of the **DiagInfoString* argument.
SQLINTEGER *	*StringLengthPtr*	Output	A pointer to a buffer in which to return the total number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function), excluding the number of bytes that are required for the null termination character, available to return in the **DiagInfoString* argument. If the number of bytes to return is greater than the value of the *BufferLength* argument, the text in the **DiagInfoString* argument is truncated to the value of the *BufferLength* argument minus the length of a null termination character. CLI then uses a null character to terminate the text in the **DiagInfoString* argument.

Usage

After you modify the db2dsdriver.cfg file, you can issue the SQLReloadConfig() function to reload the entries for the section of the db2dsdriver.cfg file that you specify in the *ConfigProperty* argument.

Currently, you can reload only the **DSD_ACR_AFFINITY** configuration property. The **DSD_ACR_AFFINITY** configuration property consists of the following parameters, which you define in the automatic client reroute (<acr>) section in the db2dsdriver.cfg file:

- <alternateserverlist>
- <affinitylist>
- <clientaffinitydefined>
- <clientaffinityroundrobin>

Modifications to other parameters in the <acr> section are ignored.

When your application issues the SQLReloadConfig() function, the reload causes reevaluation of affinity members for all connections at the *next transaction interval*. The next transaction interval refers to the next transaction boundary, which is defined when a commit or rollback occurs. This reevaluation of affinity members involves validating entries for all alternate servers within the <acr> section. For each server, an attempt is made to open a socket by using the specified host name and port number. If all servers in the alternate server list of an active database connection are unreachable, an error message is returned in the **DiagInfoString** argument of the SQLReloadConfig() function:

```
IBM DB2 [CLI Driver] <database>:<hostname>:<port> - None of the servers,
   specified under <alternateserverlist> section, are reachable.
```

Affinity for an idle connection is evaluated only when the connection becomes active. When the connection becomes active, it is moved to an affinity member.

If you modify a section of the db2dsdriver.cfg file other than the <acr> section, the SQLReloadConfig() function returns a value of SQL_ERROR in the *DiagInfoString* argument. Also, if you remove or add a <acr> section for existing database entries in the db2dsdriver.cfg file, the SQLReloadConfig() function returns an error. If the SQLReloadConfig() function returns an error, applications continue to access the old db2dsdriver.cfg file contents.

The SQLReloadConfig() function updates in-memory versions of automatic client reroute (ACR) affinities with all the databases that are listed in the current db2dsdriver.cfg file. If the function finds that the <acr> section of any database entry is invalid, the next attempt to connect to that database results in an error. The detection of invalid <acr> sections also terminates connections for active databases at the next transaction interval. Possible causes for an invalid <acr> section are as follows:

- The <database> section is missing (results in SQL_ERROR being returned).
- The <acr> section is missing (results in SQL_ERROR being returned).
- A new alternate server list is empty.
- CLI encountered internal buffer boundary limit to hold new server list.

The SQLReloadConfig() function has no default timeout value for attempting to open a socket against an alternate server port. You can set a specific timeout value by using the **SocketTimeOut** parameter in the global section of the db2dsdriver.cfg file.

Return codes

- SQL_ERROR
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

Diagnostics

The following tables provide lists of errors and warnings that can result from calling the SQLReloadConfig() function.

Table 128. SQLReloadConfig() function error messages

Error message	Explanation
IBM DB2 [CLI Driver] Another SQLReloadConfig() execution is already under progress.	The SQLReloadConfig() function was already called for a particular process and is running.
IBM DB2 [CLI Driver] Unrecognized value found in ConfigProperty argument of SQLReloadConfig().	A value other than the **DSD_ACR_AFFINITY** configuration property was specified.
IBM DB2 [CLI Driver] db2dsdriver.cfg file to be reloaded cannot be located in the expected location.	The db2dsdriver.cfg file cannot be accessed. Ensure that the db2dsdriver.cfg file is present and has global read permission.
IBM DB2 [CLI Driver] Sections other than one specified in ConfigProperty argument is found modified, which is not supported.	A section other than the <acr> section was modified.
IBM DB2 [CLI Driver] CLI subsystem is not initialized. Use SQLAllocHandle() to allocate environment handle.	You must allocate the environment handle before calling the SQLReloadConfig() function.
IBM DB2 [CLI Driver] <database>:<hostname>:<port> - Either all or one of <client>, <affinitylist>, <alternateserverlist> sections are missing in db2dsdriver.cfg.	The db2dsdriver.cfg file is missing an entry for all or one of the <client>, <affinitylist>, and <alternateserverlist> sections.
IBM DB2 [CLI Driver] <database>:<hostname>:<port> - None of the servers, specified under <alternateserverlist> section, are reachable.	The IBM driver cannot establish connection to all the servers that are specified in the <alternateserverlist> section of the db2dsdriver.cfg file.
IBM DB2 [CLI Driver] <database>:<hostname>:<port> - cannot find appropriate port number for service name <srvcname>.	The IBM driver cannot allocate a port number for the service name that is specified in the error message.

Table 129. SQLReloadConfig() function warning message

Error message	Explanation
IBM DB2 [CLI Driver] <database>:<hostname>:<port> - The following server(s) from <alternateserverlist> were unreachable "<hostname>:<port>,<hostname>:port,...".	The IBM driver cannot establish connection to some of the servers that were specified in the <alternateserverlist> section of the db2dsdriver.cfg file.

Restrictions

The **DSD_ACR_AFFINITY** configuration property of the db2dsdriver.cfg file is the only property that you can reload by using the SQLReloadConfig() function.

SQLRowCount function (CLI) - Get row count

Returns the number of rows in a table that were affected by an UPDATE, an INSERT, a DELETE, or a MERGE statement issued against the table, or a view based on the table.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

SQLRowCount() returns the number of rows in a table that were affected by an UPDATE, an INSERT, a DELETE, or a MERGE statement issued against the table, or a view based on the table.

You must call SQLExecute() or SQLExecDirect() before calling SQLRowCount().

Syntax

```
SQLRETURN   SQLRowCount (
               SQLHSTMT     StatementHandle,   /* hstmt */
               SQLLEN       *RowCountPtr);     /* pcrow */
```

Function arguments

Table 130. SQLRowCount arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLLEN *	*RowCountPtr*	output	Pointer to location where the number of rows affected is stored. If the SQL_ATTR_PARC_BATCH connection attribute is set to SQL_PARC_BATCH_ENABLE, the location size must be the size of an array. See the Usage section for details.

Usage

If the last issued statement referenced by the input statement handle was not an UPDATE, an INSERT, a DELETE, or a MERGE statement or if the statement did not run successfully, the function sets the contents of *StatementHandle* to -1.

If you use the SQLRowCount() function on a non-scrollable SELECT-only cursor, the function sets the contents of *RowCountPtr* to -1. The number of rows is not available until all of the data has been fetched. You can use the CLI statement attribute SQL_ATTR_ROWCOUNT_PREFETCH to enable the client to request the full row count before fetching the data.

Restriction: The SQL_ATTR_ROWCOUNT_PREFETCH attribute is not supported when the cursor contains LOBs or XML.

If the SQL_ATTR_PARC_BATCH connection attribute is set to SQL_PARC_BATCH_ENABLE, then the SQL_ATTR_PARAMOPT_ATOMIC attribute must be set to SQL_ATOMIC_NO and the *RowCountPtr* argument must be pointing to an array of type SQLLEN *. The length of this array must be equal to SQL_ATTR_PARAMSET_SIZE. Upon successful execution of non-atomic update, delete or insert operations, the number of rows in a table that were affected by each parameter set is stored in this array.

Any rows in other tables that might have been affected by the statement (for example, due to cascading deletes) are not included in the count.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 131. SQLRowCount SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.

Table 131. SQLRowCount SQLSTATEs (continued)

SQLSTATE	Description	Explanation
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	The function was called before calling SQLExecute() or SQLExecDirect() for the *StatementHandle*.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.

Authorization

None.

SQLSetColAttributes function (CLI) - Set column attributes

In ODBC 3.0, SQLSetColAttributes() has been deprecated, and CLI no longer supports this function.

Now that CLI uses deferred prepare by default, there is no need for the functionality of SQLSetColAttributes().

SQLSetConnectAttr function (CLI) - Set connection attributes

Sets attributes that govern aspects of connections.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLSetConnectAttrW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLSetConnectAttr (
               SQLHDBC          ConnectionHandle,  /* hdbc */
               SQLINTEGER       Attribute,         /* fOption */
               SQLPOINTER       ValuePtr,          /* pvParam */
               SQLINTEGER       StringLength);     /* fStrLen */
```

Function arguments

Table 132. SQLSetConnectAttr arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	Connection handle.
SQLINTEGER	*Attribute*	input	Attribute to set, listed in the connection attributes list.

Table 132. SQLSetConnectAttr arguments (continued)

Data type	Argument	Use	Description
SQLPOINTER	*ValuePtr*	input	Pointer to the value to be associated with *Attribute*. Depending on the value of *Attribute*, *ValuePtr* will be a 32-bit unsigned integer value or pointer to a null-terminated character string. Note that if the *Attribute* argument is a driver-specific value, the value in **ValuePtr* can be a signed integer. Refer to the connection attributes list for details.
SQLINTEGER	*StringLength*	input	If *Attribute* is an ODBC-defined attribute and *ValuePtr* points to a character string or a binary buffer, this argument should be the length of **ValuePtr*. For character string data, *StringLength* should contain the number of bytes in the string. If *Attribute* is an ODBC-defined attribute and *ValuePtr* is an integer, *StringLength* is ignored. If *Attribute* is a CLI attribute, the application indicates the nature of the attribute by setting the *StringLength* argument. *StringLength* can have the following values: • If *ValuePtr* is a pointer to a character string, then *StringLength* is the number of bytes needed to store the string or SQL_NTS. • If *ValuePtr* is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in *StringLength*. This places a negative value in *StringLength*. • If *ValuePtr* is a pointer to a value other than a character string or a binary string, then *StringLength* should have the value SQL_IS_POINTER. • If *ValuePtr* contains a fixed-length value, then *StringLength* is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.

Usage

Setting statement attributes using SQLSetConnectAttr() no longer supported

The ability to set statement attributes using SQLSetConnectAttr() is no longer supported. To support applications written before version 5, some statement attributes can be set using SQLSetConnectAttr() in this release of CLI. All applications that rely on this behavior, however, should be updated to use `SQLSetStmtAttr()` instead.

If SQLSetConnectAttr() is called to set a statement attribute that sets the header field of a descriptor, the descriptor field is set for the application descriptors currently associated with all statements on the connection. However, the attribute setting does not affect any descriptors that might be associated with the statements on that connection in the future.

Connection Attributes

At any time between allocating and freeing a connection, an application can call SQLSetConnectAttr(). All connection and statement attributes successfully set by the application for the connection persist until `SQLFreeHandle()` is called on the connection.

Some connection attributes can be set only before a connection has been made; others can be set only after a connection has been made, while some cannot be set once a statement is allocated. Refer to the connection attributes list for details on when each attribute can be set.

Some connection attributes support substitution of a similar value if the data source does not support the value specified in *ValuePtr*. In such cases, CLI returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed.). To determine the substituted value, an application calls `SQLGetConnectAttr()`.

The format of information set through *ValuePtr* depends on the specified *Attribute*. SQLSetConnectAttr() will accept attribute information in one of two different formats: a null-terminated character string or a 32-bit integer value. The format of each is noted in the attribute's description. Character strings pointed to by the *ValuePtr* argument of SQLSetConnectAttr() have a length of *StringLength* bytes. The *StringLength* argument is ignored if the length is defined by the attribute.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

CLI can return SQL_SUCCESS_WITH_INFO to provide information about the result of setting an option.

When *Attribute* is a statement attribute, SQLSetConnectAttr() can return any SQLSTATEs returned by `SQLSetStmtAttr()`.

Table 133. SQLSetConnectAttr SQLSTATEs

SQLSTATE	Description	Explanation
01000	General error.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed.	CLI did not support the value specified in **ValuePtr* and substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)
08002	Connection in use.	The argument *Attribute* was SQL_ATTR_ODBC_CURSORS and CLI was already connected to the data source.
08003	Connection is closed.	An *Attribute* value was specified that required an open connection, but the *ConnectionHandle* was not in a connected state.
08S01	Communication link failure.	The communication link between CLI and the data source to which it was connected failed before the function completed processing.
24000	Invalid cursor state.	The argument *Attribute* was SQL_ATTR_CURRENT_QUALIFIER and a result set was pending.

Table 133. SQLSetConnectAttr SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY000	General error.	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by `SQLGetDiagRec()` in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY009	Invalid argument value.	A null pointer was passed for *ValuePtr* and the value in **ValuePtr* was a string value.
HY010	Function sequence error.	An asynchronously executing function was called for a *StatementHandle* associated with the *ConnectionHandle* and was still executing when SQLSetConnectAttr() was called. `SQLExecute()` or `SQLExecDirect()` was called for a *StatementHandle* associated with the *ConnectionHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. `SQLBrowseConnect()` was called for the *ConnectionHandle* and returned SQL_NEED_DATA. This function was called before `SQLBrowseConnect()` returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.
HY011	Operation invalid at this time.	The argument *Attribute* was SQL_ATTR_TXN_ISOLATION and a transaction was open.
HY024	Invalid attribute value.	Given the specified *Attribute* value, an invalid value was specified in **ValuePtr*. (CLI returns this SQLSTATE only for connection and statement attributes that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE. For all other connection and statement attributes, CLI must verify the value specified in *ValuePtr*.) The *Attribute* argument was SQL_ATTR_TRACEFILE or SQL_ATTR_TRANSLATE_LIB, and **ValuePtr* was an empty string.
HY090	Invalid string or buffer length.	The *StringLength* argument was less than 0, but was not SQL_NTS.
HY092	Option type out of range.	The value specified for the argument *Attribute* was not valid for this version of CLI.
HYC00	Driver not capable.	The value specified for the argument *Attribute* was a valid connection or statement attribute for the version of the CLI driver, but was not supported by the data source.

Restrictions

None.

Example

```
/* set AUTOCOMMIT on */
cliRC = SQLSetConnectAttr(hdbc,
                          SQL_ATTR_AUTOCOMMIT,
                          (SQLPOINTER)SQL_AUTOCOMMIT_ON,
                          SQL_NTS);
/* ... */
```

```
/* set AUTOCOMMIT OFF */
cliRC = SQLSetConnectAttr(hdbc,
                          SQL_ATTR_AUTOCOMMIT,
                          (SQLPOINTER)SQL_AUTOCOMMIT_OFF,
                          SQL_NTS);
```

SQLSetConnection function (CLI) - Set connection handle

This function is needed if the application needs to deterministically switch to a particular connection before continuing execution. It should only be used when the application is mixing CLI function calls with embedded SQL function calls and where multiple connections are used.

Specification:

- **CLI** 2.1

Syntax

```
SQLRETURN   SQLSetConnection (SQLHDBC          ConnectionHandle); /* hdbc */
```

Function arguments

Table 134. SQLSetConnection arguments

Data type	Argument	Use	Description
SQLHDBC	*ConnectionHandle*	input	The connection handle associated with the connection that the application wishes to switch to.

Usage

In CLI version 1 it was possible to mix CLI calls with calls to routines containing embedded SQL as long as the connect request was issued via the CLI connect function. The embedded SQL routine would simply use the existing CLI connection.

Although this is still true, there is a potential complication: CLI allows multiple concurrent connections. This means that it is no longer clear which connection an embedded SQL routine would use upon being invoked. In practice, the embedded routine would use the connection associated with the most recent network activity. However, from the application's perspective, this is not always deterministic and it is difficult to keep track of this information. SQLSetConnection() is used to allow the application to *explicitly* specify which connection is active. The application can then call the embedded SQL routine.

SQLSetConnection() is not needed if the application makes use of CLI exclusively. Under those conditions, each statement handle is implicitly associated with a connection handle and there is never any confusion as to which connection a particular CLI function applies.

Return codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 135. SQLSetConnection SQLSTATEs

SQLSTATE	Description	Explanation
08003	Connection is closed.	The connection handle provided is not currently associated with an open connection to a database server.
HY000	General error.	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by `SQLGetDiagRec()` in the argument *MessageText* describes the error and its cause.

Restrictions

None.

Example

```
/* perform statements on the first connection */
cliRC = SQLSetConnection(hdbc1);

/* ... */

/* perform statements on the second connection */
cliRC = SQLSetConnection(hdbc2);
```

SQLSetConnectOption function (CLI) - Set connection option

In ODBC 3.0, SQLSetConnectOption() has been deprecated and replaced with SQLSetConnectAttr().

Although this version of CLI continues to support SQLSetConnectOption(), use SQLSetConnectAttr() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

Note:

This deprecated function cannot be used in a 64-bit environment.

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLSetConnectOptionW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

The statement:

```
SQLSetConnectOption(
           hdbc,
           SQL_AUTOCOMMIT,
           SQL_AUTOCOMMIT_OFF);
```

for example, would be rewritten using the new function as:

```
SQLSetConnectAttr(
           hdbc,
           SQL_ATTR_AUTOCOMMIT,
           SQL_AUTOCOMMIT_OFF,
           0);
```

SQLSetCursorName function (CLI) - Set cursor name

Associates a cursor name with the statement handle.

This function is optional because CLI implicitly generates a cursor name. The implicit cursor name is available after the dynamic SQL has been prepared on the statement handle.

Specification:

- **CLI** 1.1
- **ODBC** 1.0
- **ISO CLI**

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLSetCursorNameW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLSetCursorName (
               SQLHSTMT          StatementHandle,  /* hstmt */
               SQLCHAR           *CursorName,      /* szCursor */
               SQLSMALLINT       NameLength);      /* cbCursor */
```

Function arguments

Table 136. SQLSetCursorName arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle
SQLCHAR *	*CursorName*	input	Cursor name
SQLSMALLINT	*NameLength*	input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store the *CursorName* argument.

Usage

CLI always generates and uses an internally generated cursor name when a query is prepared or executed directly. SQLSetCursorName() allows an application-defined cursor name to be used in an SQL statement (a positioned UPDATE or DELETE). CLI maps this name to the internal name. The name will remain associated with the statement handle, until the handle is dropped, or another `SQLSetCursorName()` is called on this statement handle.

Although `SQLGetCursorName()` will return the name set by the application (if one was set), error messages associated with positioned UPDATE and DELETE statements will refer to the internal name. For this reason, do not use SQLSetCursorName() for positioned UPDATEs and DELETEs, but instead use the internal name which can be obtained by calling `SQLGetCursorName()`.

Cursor names must follow these rules:

- All cursor names within the connection must be unique.

- Each cursor name must be less than or equal to 128 bytes in length. Any attempt to set a cursor name longer than 128 bytes results in truncation of that cursor name to 128 bytes. (No warning is generated.)
- Since internally generated names begin with SQLCUR or SQL_CUR, the application must not input a cursor name starting with either SQLCUR or SQL_CUR in order to avoid conflicts with internal names.
- Since a cursor name is considered an identifier in SQL, it must begin with an English letter (a-z, A-Z) followed by any combination of digits (0-9), English letters or the underscore character (_).
- To permit cursor names containing characters other than those listed (such as National Language Set or Double Bytes Character Set characters), the application must enclose the cursor name in double quotation marks (").
- Unless the input cursor name is enclosed in double quotation marks, all leading and trailing blanks from the input cursor name string will be removed.

For efficient processing, applications should not include any leading or trailing spaces in the *CursorName* buffer. If the *CursorName* buffer contains a delimited identifier, applications should position the first double quotation marks as the first character in the *CursorName* buffer.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 137. SQLSetCursorName SQLSTATEs

SQLSTATE	Description	Explanation
34000	Invalid cursor name.	The cursor name specified by the argument *CursorName* was invalid. The cursor name either begins with "SQLCUR" or "SQL_CUR" or violates the cursor naming rules (Must begin with a-z or A-Z followed by any combination of English letters, digits, or the '_' character. The cursor name specified by the argument *CursorName* already exists. The cursor name length is greater than the value returned by `SQLGetInfo()` with the SQL_MAX_CURSOR_NAME_LEN argument.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
58004	Unexpected system failure.	Unrecoverable system error.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY009	Invalid argument value.	*CursorName* was a null pointer.

Table 137. SQLSetCursorName SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	There is an open or positioned cursor on the statement handle. The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY090	Invalid string or buffer length.	The argument *NameLength* was less than **0**, but not equal to SQL_NTS.

Authorization

None.

Example

```
/* set the name of the cursor */
rc = SQLSetCursorName(hstmtSelect, (SQLCHAR *)"CURSNAME", SQL_NTS);
```

SQLSetDescField function (CLI) - Set a single field of a descriptor record

Sets the value of a single field of a descriptor record.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLSetDescFieldW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLSetDescField  (SQLHDESC          DescriptorHandle,
                              SQLSMALLINT       RecNumber,
                              SQLSMALLINT       FieldIdentifier,
                              SQLPOINTER        ValuePtr,
                              SQLINTEGER        BufferLength);
```

Function arguments

Table 138. SQLSetDescField arguments

Data type	Argument	Use	Description
SQLHDESC	*DescriptorHandle*	input	Descriptor handle.
SQLSMALLINT	*RecNumber*	input	Indicates the descriptor record containing the field that the application seeks to set. Descriptor records are numbered from 0, with record number 0 being the bookmark record. The *RecNumber* argument is ignored for header fields.
SQLSMALLINT	*FieldIdentifier*	input	Indicates the field of the descriptor whose value is to be set. For more information, refer to the list of values for the descriptor *FieldIdentifier* argument.
SQLPOINTER	*ValuePtr*	input	Pointer to a buffer containing the descriptor information, or a four-byte value. The data type depends on the value of *FieldIdentifier*. If *ValuePtr* is a four-byte value, either all four of the bytes are used, or just two of the four are used, depending on the value of the *FieldIdentifier* argument.
SQLINTEGER	*BufferLength*	input	If *FieldIdentifier* is an ODBC-defined field and *ValuePtr* points to a character string or a binary buffer, this argument should be the length of **ValuePtr*. For character string data, *BufferLength* should contain the number of bytes in the string. If *FieldIdentifier* is an ODBC-defined field and *ValuePtr* is an integer, *BufferLength* is ignored. If *FieldIdentifier* is a driver-defined field, the application indicates the nature of the field by setting the *BufferLength* argument. *BufferLength* can have the following values: • If *ValuePtr* is a pointer to a character string, then *BufferLength* is the number of bytes needed to store the string or SQL_NTS. • If *ValuePtr* is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in *BufferLength* This places a negative value in *BufferLength*. • If *ValuePtr* is a pointer to a value other than a character string or a binary string, then *BufferLength* should have the value SQL_IS_POINTER. • If *ValuePtr* contains a fixed-length value, then *BufferLength* is either SQL_IS_INTEGER, SQL_IS_UINTEGER, SQL_IS_SMALLINT, or SQL_IS_USMALLINT, as appropriate.

Usage

An application can call SQLSetDescField() to set any descriptor field one at a time. One call to SQLSetDescField() sets a single field in a single descriptor. This function can be called to set any field in any descriptor type, provided the field can be set. See the descriptor header and record field initialization values for more information.

Note: If a call to SQLSetDescField() fails, the contents of the descriptor record identified by the *RecNumber* argument are undefined.

Other functions can be called to set multiple descriptor fields with a single call of the function. The SQLSetDescRec() function sets a variety of fields that affect the data type and buffer bound to a column or parameter (the SQL_DESC_TYPE, SQL_DESC_DATETIME_INTERVAL_CODE, SQL_DESC_OCTET_LENGTH, SQL_DESC_PRECISION, SQL_DESC_SCALE, SQL_DESC_DATA_PTR, SQL_DESC_OCTET_LENGTH_PTR, and SQL_DESC_INDICATOR_PTR fields). SQLBindCol() or SQLBindParameter() can be used to make a complete specification for the binding of a column or parameter. These functions each set a specific group of descriptor fields with one function call.

SQLSetDescField() can be called to change the binding buffers by adding an offset to the binding pointers (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, or SQL_DESC_OCTET_LENGTH_PTR). This changes the binding buffers without calling SQLBindCol() or SQLBindParameter(). This allows an application to quickly change SQL_DESC_DATA_PTR without concern for changing other fields, for example, SQL_DESC_DATA_TYPE.

Descriptor header fields are set by calling SQLSetDescField() with a *RecNumber* of 0, and the appropriate *FieldIdentifier*. Many header fields contain statement attributes, so can also be set by a call to SQLSetStmtAttr(). This allows applications to set a statement attribute without first obtaining a descriptor handle. A *RecNumber* of 0 is also used to set bookmark fields.

Note: The statement attribute SQL_ATTR_USE_BOOKMARKS should always be set before calling SQLSetDescField() to set bookmark fields. While this is not mandatory, it is strongly recommended.

Sequence of setting descriptor fields

When setting descriptor fields by calling SQLSetDescField(), the application must follow a specific sequence:

- The application must first set the SQL_DESC_TYPE, SQL_DESC_CONCISE_TYPE, or SQL_DESC_DATETIME_INTERVAL_CODE field.

  Note: SQL_DESC_DATETIME_INTERVAL_CODE is defined by ODBC but not supported by CLI.
- After one of these fields has been set, the application can set an attribute of a data type, and the driver sets data type attribute fields to the appropriate default values for the data type. Automatic defaulting of type attribute fields ensures that the descriptor is always ready to use once the application has specified a data type. If the application explicitly sets a data type attribute, it is overriding the default attribute.
- After one of the fields listed in Step 1 has been set, and data type attributes have been set, the application can set SQL_DESC_DATA_PTR. This prompts a consistency check of descriptor fields. If the application changes the data type or attributes after setting the SQL_DESC_DATA_PTR field, then the driver sets SQL_DESC_DATA_PTR to a null pointer, unbinding the record. This forces the application to complete the proper steps in sequence, before the descriptor record is usable.

Initialization of descriptor fields

When a descriptor is allocated, the fields in the descriptor can be initialized to a default value, be initialized without a default value, or be undefined for the type of descriptor. Refer to the list of descriptor header and record field initialization values for details.

The fields of an IRD have a default value only after the statement has been prepared or executed and the IRD has been populated, not when the statement handle or descriptor has been allocated. Until the IRD has been populated, any attempt to gain access to a field of an IRD will return an error.

Some descriptor fields are defined for one or more, but not all, of the descriptor types (ARDs and IRDs, and APDs and IPDs). When a field is undefined for a type of descriptor, it is not needed by any of the functions that use that descriptor. Because a descriptor is a logical view of data, rather than an actual data structure, these extra fields have no effect on the defined fields.

The fields that can be accessed by SQLGetDescField() are not necessarily set by SQLSetDescField(). Fields that can be set by SQLSetDescField() are described in the descriptor header and record field initialization values list.

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 139. SQLSetDescField SQLSTATEs

SQLSTATE	Description	Explanation
01000	General warning	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed.	CLI did not support the value specified in **ValuePtr* (if *ValuePtr* was a pointer) or the value in *ValuePtr* (if *ValuePtr* was a four-byte value), or **ValuePtr* was invalid because of SQL constraints or requirements, so CLI substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)
07009	Invalid descriptor index.	The *FieldIdentifier* argument was a header field, and the *RecNumber* argument was not 0. The *RecNumber* argument was 0 and the *DescriptorHandle* was an IPD. The *RecNumber* argument was less than 0.
08S01	Communication link failure.	The communication link between CLI and the data source to which it was connected failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.

Table 139. SQLSetDescField SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	The *DescriptorHandle* was associated with a StatementHandle for which an asynchronously executing function (not this one) was called and was still executing when this function was called. SQLExecute() or SQLExecDirect() was called for the *StatementHandle* with which the *DescriptorHandle* was associated and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY016	Cannot modify an implementation row descriptor.	The *DescriptorHandle* argument was associated with an IRD, and the *FieldIdentifier* argument was not SQL_DESC_ARRAY_STATUS_PTR.
HY021	Inconsistent descriptor information.	The TYPE field, or any other field associated with the TYPE field in the descriptor, was not valid or consistent. The TYPE field was not a valid CLI C type. Descriptor information checked during a consistency check was not consistent.
HY091	Invalid descriptor field identifier.	The value specified for the *FieldIdentifier* argument was not a CLI defined field and was not a defined value. The value specified for the *RecNumber* argument was greater than the value in the SQL_DESC_COUNT field. The *FieldIdentifier* argument was SQL_DESC_ALLOC_TYPE.
HY092	Option type out of range.	The value specified for the *Attribute* argument was not valid.
HY094	Invalid scale value.	The value specified for *pfParamType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified for *DecimalDigits* was less than 0 or greater than the value for the argument *pcbColDef* (precision). The value specified for *pfParamType* was SQL_C_TYPE_TIMESTAMP and the value for *pfParamType* was either SQL_CHAR or SQL_VARCHAR and the value for *DecimalDigits* was less than 0 or greater than 9. The value specified for *pfParamType* was SQL_C_TIMESTAMP_EXT and the value for DecimalDigits was less than 0 or greater than 12.
HY105	Invalid parameter type.	The value specified for the SQL_DESC_PARAMETER_TYPE field was invalid. (For more information, see the *InputOutputType* Argument section in SQLBindParameter().)

Restrictions

None.

Example

```
/* set a single field of a descriptor record */
rc = SQLSetDescField(hARD,
                     1,
                     SQL_DESC_TYPE,
                     (SQLPOINTER)SQL_SMALLINT,
                     SQL_IS_SMALLINT);
```

SQLSetDescRec function (CLI) - Set multiple descriptor fields for a column or parameter data

The SQLSetDescRec() function sets multiple descriptor fields that affect the data type and buffer bound to a column or parameter data.

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Syntax

```
SQLRETURN   SQLSetDescRec    (SQLHDESC          DescriptorHandle,
                              SQLSMALLINT       RecNumber,
                              SQLSMALLINT       Type,
                              SQLSMALLINT       SubType,
                              SQLLEN            Length,
                              SQLSMALLINT       Precision,
                              SQLSMALLINT       Scale,
                              SQLPOINTER        DataPtr,
                              SQLLEN            *StringLengthPtr,
                              SQLLEN            *IndicatorPtr);
```

Function arguments

Table 140. SQLSetDescRec arguments

Data type	Argument	Use	Description
SQLHDESC	*DescriptorHandle*	input	Descriptor handle. This must not be an IRD handle.
SQLSMALLINT	*RecNumber*	input	Indicates the descriptor record that contains the fields to be set. Descriptor records are numbered from 0, with record number 0 being the bookmark record. This argument must be equal to or greater than 0. If *RecNumber* is greater than the value of SQL_DESC_COUNT, SQL_DESC_COUNT is changed to the value of *RecNumber*.
SQLSMALLINT	*Type*	input	The value to which to set the SQL_DESC_TYPE field for the descriptor record.
SQLSMALLINT	*SubType*	input	For records whose type is SQL_DATETIME, this is the value to which to set the SQL_DESC_DATETIME_INTERVAL_CODE field.
SQLLEN	*Length*	input	The value to which to set the SQL_DESC_OCTET_LENGTH field for the descriptor record.
SQLSMALLINT	*Precision*	input	The value to which to set the SQL_DESC_PRECISION field for the descriptor record.

Table 140. SQLSetDescRec arguments (continued)

Data type	Argument	Use	Description
SQLSMALLINT	*Scale*	input	The value to which to set the SQL_DESC_SCALE field for the descriptor record.
SQLPOINTER	*DataPtr*	Deferred Input or Output	The value to which to set the SQL_DESC_DATA_PTR field for the descriptor record. *DataPtr* can be set to a null pointer to set the SQL_DESC_DATA_PTR field to a null pointer.
SQLLEN *	*StringLengthPtr*	Deferred Input or Output	The value to which to set the SQL_DESC_OCTET_LENGTH_PTR field for the descriptor record. *StringLengthPtr* can be set to a null pointer to set the SQL_DESC_OCTET_LENGTH_PTR field to a null pointer.
SQLLEN *	*IndicatorPtr*	Deferred Input or Output	The value to which to set the SQL_DESC_INDICATOR_PTR field for the descriptor record. *IndicatorPtr* can be set to a null pointer to set the SQL_DESC_INDICATOR_PTR field to a null pointer.

Usage

An application can call SQLSetDescRec() to set the following fields for a single column or parameter:
- SQL_DESC_TYPE
- SQL_DESC_OCTET_LENGTH
- SQL_DESC_PRECISION
- SQL_DESC_SCALE
- SQL_DESC_DATA_PTR
- SQL_DESC_OCTET_LENGTH_PTR
- SQL_DESC_INDICATOR_PTR

SQL_DESC_DATETIME_INTERVAL_CODE can only be updated if SQL_DESC_TYPE indicates SQL_DATETIME.

Note: If a call to SQLSetDescRec() fails, the contents of the descriptor record identified by the *RecNumber* argument are undefined.

When binding a column or parameter, SQLSetDescRec() allows you to change multiple fields affecting the binding without calling SQLBindCol() or SQLBindParameter(), or making multiple calls to SQLSetDescField(). SQLSetDescRec() can set fields on a descriptor not currently associated with a statement. Note that SQLBindParameter() sets more fields than SQLSetDescRec(), can set fields on both an APD and an IPD in one call, and does not require a descriptor handle.

The statement attribute SQL_ATTR_USE_BOOKMARKS should always be set before calling SQLSetDescRec() with a *RecNumber* argument of 0 to set bookmark fields. While this is not mandatory, it is strongly recommended.

Return Codes
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 141. SQLSetDescRec SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
07009	Invalid descriptor index.	The *RecNumber* argument was set to 0, and the *DescriptorHandle* was an IPD handle. The *RecNumber* argument was less than 0. The *RecNumber* argument was greater than the maximum number of columns or parameters that the data source can support, and the *DescriptorHandle* argument was an APD, IPD, or ARD. The *RecNumber* argument was equal to 0, and the *DescriptorHandle* argument referred to an implicitly allocated APD. (This error does not occur with an explicitly allocated application descriptor, because it is not known whether an explicitly allocated application descriptor is an APD or ARD until execute time.)
08S01	Communication link failure.	The communication link between CLI and the data source to which it was connected failed before the function completed processing.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY010	Function sequence error.	The *DescriptorHandle* was associated with a *StatementHandle* for which an asynchronously executing function (not this one) was called and was still executing when this function was called. SQLExecute() or SQLExecDirect() was called for the *StatementHandle* with which the *DescriptorHandle* was associated and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters.
HY013	Unexpected memory handling error.	DB2 CLI was unable to access memory required to support execution or completion of the function.
HY016	Cannot modify an implementation row descriptor.	The *DescriptorHandle* argument was associated with an IRD.
HY021	Inconsistent descriptor information.	The *Type* field, or any other field associated with the TYPE field in the descriptor, was not valid or consistent. Descriptor information checked during a consistency check was not consistent.

Table 141. SQLSetDescRec SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY094	Invalid scale value.	The value specified for *pfParamType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified for *DecimalDigits* was less than 0 or greater than the value for the argument *pcbColDef* (precision). The value specified for *pfParamType* was SQL_C_TYPE_TIMESTAMP and the value for *pfParamType* was either SQL_CHAR or SQL_VARCHAR and the value for *DecimalDigits* was less than 0 or greater than 9. The value specified for *pfParamType* was SQL_C_TIMESTAMP_EXT and the value for DecimalDigits was less than 0 or greater than 12.

Restrictions

None.

Example

```
SQLSMALLINT type;
SQLINTEGER length, datalen;
SQLSMALLINT id_no;
/* ... */

/* set multiple descriptor fields for a column or parameter data */
rc = SQLSetDescRec(hARD, 1, type, 0, length, 0, 0, &id_no, &datalen, NULL);
```

SQLSetEnvAttr function (CLI) - Set environment attribute

`SQLSetEnvAttr()` sets an environment attribute for the current environment.

Specification:

- **CLI** 2.1
- **ISO CLI**

Syntax

```
SQLRETURN   SQLSetEnvAttr    (SQLHENV          EnvironmentHandle, /* henv */
                              SQLINTEGER       Attribute,
                              SQLPOINTER       ValuePtr,          /* Value */
                              SQLINTEGER       StringLength);
```

Function arguments

Table 142. SQLSetEnvAttr arguments

Data type	Argument	Use	Description
SQLHENV	*EnvironmentHandle*	Input	Environment handle.
SQLINTEGER	*Attribute*	Input	Environment attribute to set; refer to the list of CLI environment attributes for descriptions.
SQLPOINTER	*ValuePtr*	Input	Value for the *Attribute*.
SQLINTEGER	*StringLength*	Input	Length of *ValuePtr* in bytes if the attribute value is a character string; if *Attribute* does not denote a string, then CLI ignores *StringLength*.

Usage

Once set, the attribute's value affects all connections under this environment.

The application can obtain the current attribute value by calling SQLGetEnvAttr().

Refer to the list of CLI environment attributes for the attributes that can be set with SQLSetEnvAttr().

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 143. SQLSetEnvAttr SQLSTATEs

SQLSTATE	Description	Explanation
HY011	Operation invalid at this time.	Applications cannot set environment attributes while connection handles are allocated on the environment handle.
HY024	Invalid attribute value	Given the specified *Attribute* value, an invalid value was specified in **ValuePtr*.
HY090	Invalid string or buffer length	The *StringLength* argument was less than 0, but was not SQL_NTS.
HY092	Option type out of range.	An invalid *Attribute* value was specified.
HYC00	Driver not capable.	The specified *Attribute* is not supported by CLI. Given specified *Attribute* value, the value specified for the argument *ValuePtr* is not supported.

Restrictions

None.

Example

```
/* set environment attribute */
cliRC = SQLSetEnvAttr(henv, SQL_ATTR_OUTPUT_NTS, (SQLPOINTER) SQL_TRUE, 0);
```

SQLSetParam function (CLI) - Bind a parameter marker to a buffer or LOB locator

In ODBC 2.0 and later, SQLSetParam() is deprecated and replaced with SQLBindParameter().

Although this version of CLI continues to support SQLSetParam(), use SQLBindParameter() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

The CLI function SQLBindParameter() is functionally the same as the SQLSetParam() function. Both take a similar number and type of arguments,

behave the same, and return the same return codes. The difference is that SQLSetParam() does not have the *InputOutputType* or *BufferLength* arguments to specify the parameter type and maximum buffer length. Calling SQLSetParam() is functionally equivalent to calling SQLBindParameter() with the *InputOutputType* argument set to SQL_PARAM_INPUT and the *BufferLength* argument set to SQL_SETPARAM_VALUE_MAX.

The statement:

```
SQLSetParam(hstmt, 1, SQL_C_SHORT, SQL_SMALLINT, 0, 0,
            &parameter1, NULL);
```

for example, would be rewritten using the new function as:

```
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SHORT,
                 SQL_SMALLINT, 0, 0, &parameter1,
                 SQL_SETPARAM_VALUE_MAX, NULL);
```

SQLSetPos function (CLI) - Set the cursor position in a rowset

Sets the cursor position in a rowset.

Specification:

- **CLI** 5.0
- **ODBC** 1

Syntax

```
SQLRETURN   SQLSetPos (
               SQLHSTMT          StatementHandle,  /* hstmt */
               SQLSETPOSIROW     RowNumber,        /* irow */
               SQLUSMALLINT      Operation,        /* fOption */
               SQLUSMALLINT      LockType);        /* fLock */
```

Function arguments

Table 144. SQLSetPos arguments

Data type	Argument	Use	Description
SQLHSTMT	StatementHandle	input	Statement handle.
SQLSETPOSIROW	*RowNumber*	input	Position of the row in the rowset on which to perform the operation specified with the *Operation* argument. If *RowNumber* is 0, the operation applies to every row in the rowset. For additional information, see RowNumber argument.

Table 144. SQLSetPos arguments (continued)

Data type	Argument	Use	Description
SQLUSMALLINT	*Operation*	input	Operation to perform: • SQL_POSITION • SQL_REFRESH • SQL_UPDATE • SQL_DELETE • SQL_ADD ODBC also specifies the following operations for backwards compatibility only, which CLI also supports: • SQL_ADD While CLI does support SQL_ADD in SQLSetPos() calls, it is suggested that you use `SQLBulkOperations()` with the *Operation* argument set to SQL_ADD.
SQLUSMALLINT	*LockType*	input	Specifies how to lock the row after performing the operation specified in the *Operation* argument. • SQL_LOCK_NO_CHANGE ODBC also specifies the following operations which CLI does not support: • SQL_LOCK_EXCLUSIVE • SQL_LOCK_UNLOCK For additional information, see LockType argument.

Usage

RowNumber argument

The *RowNumber* argument specifies the number of the row in the rowset on which to perform the operation specified by the *Operation* argument. If *RowNumber* is 0, the operation applies to every row in the rowset. *RowNumber* must be a value from 0 to the number of rows in the rowset.

Note In the C language, arrays are 0-based, while the *RowNumber* argument is 1-based. For example, to update the fifth row of the rowset, an application modifies the rowset buffers at array index 4, but specifies a *RowNumber* of 5.

All operations position the cursor on the row specified by *RowNumber*. The following operations require a cursor position:
- Positioned update and delete statements.
- Calls to `SQLGetData()`.
- Calls to SQLSetPos() with the SQL_DELETE, SQL_REFRESH, and SQL_UPDATE options.

An application can specify a cursor position when it calls SQLSetPos(). Generally, it calls SQLSetPos() with the SQL_POSITION or SQL_REFRESH operation to position the cursor before executing a positioned update or delete statement or calling `SQLGetData()`.

Operation argument

To determine which options are supported by a data source, an application calls `SQLGetInfo()` with one of the following information types, depending on the type of cursor:

- SQL_DYNAMIC_CURSOR_ATTRIBUTES1
- SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1
- SQL_KEYSET_CURSOR_ATTRIBUTES1
- SQL_STATIC_CURSOR_ATTRIBUTES1

SQL_POSITION

CLI positions the cursor on the row specified by *RowNumber*.

The contents of the row status array pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute are ignored for the SQL_POSITION *Operation*.

SQL_REFRESH

CLI positions the cursor on the row specified by *RowNumber* and refreshes data in the rowset buffers for that row. For more information about how CLI returns data in the rowset buffers, see the descriptions of row-wise and column-wise binding.

SQLSetPos() with an *Operation* of SQL_REFRESH simply updates the status and content of the rows within the current fetched rowset. This includes refreshing the bookmarks. The data in the buffers is refreshed, but not refetched, so the membership in the rowset is fixed.

A successful refresh with SQLSetPos() will not change a row status of SQL_ROW_DELETED. Deleted rows within the rowset will continue to be marked as deleted until the next fetch. The rows will disappear at the next fetch if the cursor supports packing (in which case a subsequent `SQLFetch()` or `SQLFetchScroll()` does not return deleted rows).

A successful refresh with SQLSetPos() will change a row status of SQL_ROW_ADDED to SQL_ROW_SUCCESS (if the row status array exists).

A refresh with SQLSetPos() will change a row status of SQL_ROW_UPDATED to the row's new status (if the row status array exists).

If an error occurs in a SQLSetPos() operation on a row, the row status is set to SQL_ROW_ERROR (if the row status array exists).

For a cursor opened with a SQL_ATTR_CONCURRENCY statement attribute of SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES, a refresh with SQLSetPos() will update the optimistic concurrency values used by the data source to detect that the row has changed. This occurs for each row that is refreshed.

The contents of the row status array are ignored for the SQL_REFRESH *Operation*.

SQL_UPDATE

CLI positions the cursor on the row specified by *RowNumber* and updates the underlying row of data with the values in the rowset buffers (the *TargetValuePtr* argument in `SQLBindCol()`). It retrieves the lengths of the data from the length/indicator buffers (the *StrLen_or_IndPtr* argument in

SQLBindCol()). If the length of any column is SQL_COLUMN_IGNORE, the column is not updated. After updating the row, the corresponding element of the row status array is updated to SQL_ROW_UPDATED or SQL_ROW_SUCCESS_WITH_INFO (if the row status array exists).

The row operation array pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute can be used to indicate that a row in the current rowset should be ignored during a bulk update. For more information, see Status and operation arrays.

SQL_DELETE

CLI positions the cursor on the row specified by *RowNumber* and deletes the underlying row of data. It changes the corresponding element of the row status array to SQL_ROW_DELETED. After the row has been deleted, the following operations are not valid for the row:

- positioned update and delete statements
- calls to SQLGetData()
- calls to SQLSetPos() with *Operation* set to anything except SQL_POSITION.

Deleted rows remain visible to static and keyset-driven cursors; however, the entry in the implementation row status array (pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute) for the deleted row is changed to SQL_ROW_DELETED.

The row operation array pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute can be used to indicate that a row in the current rowset should be ignored during a bulk delete. For more information, see Status and operation arrays.

SQL_ADD

ODBC also specifies the SQL_ADD *Operation* for backwards compatibility only, which CLI also supports. It is suggested, however, that you use SQLBulkOperations() with the *Operation* argument set to SQL_ADD.

LockType argument

The *LockType* argument provides a way for applications to control concurrency. Generally, data sources that support concurrency levels and transactions will only support the SQL_LOCK_NO_CHANGE value of the *LockType* argument.

Although the *LockType* argument is specified for a single statement, the lock accords the same privileges to all statements on the connection. In particular, a lock that is acquired by one statement on a connection can be unlocked by a different statement on the same connection.

ODBC defines the following *LockType* arguments. CLI supports SQL_LOCK_NO_CHANGE. To determine which locks are supported by a data source, an application calls SQLGetInfo() with the SQL_LOCK_TYPES information type.

Table 145. Operation values

LockType argument	Lock type
SQL_LOCK_NO_CHANGE	Ensures that the row is in the same locked or unlocked state as it was before SQLSetPos() was called. This value of *LockType* allows data sources that do not support explicit row-level locking to use whatever locking is required by the current concurrency and transaction isolation levels.

Table 145. Operation values (continued)

LockType argument	Lock type
SQL_LOCK_EXCLUSIVE	Not supported by CLI. Locks the row exclusively.
SQL_LOCK_UNLOCK	Not supported by CLI. Unlocks the row.

Status and operation arrays

The following status and operation arrays are used when calling SQLSetPos():

- The row status array (as pointed to by the SQL_DESC_ARRAY_STATUS_PTR field in the IRD and the SQL_ATTR_ROW_STATUS_ARRAY statement attribute) contains status values for each row of data in the rowset. The status values are set in this array after a call to `SQLFetch()`, `SQLFetchScroll()`, or `SQLSetPos()`. This array is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute.
- The row operation array (as pointed to by the SQL_DESC_ARRAY_STATUS_PTR field in the ARD and the SQL_ATTR_ROW_OPERATION_ARRAY statement attribute) contains a value for each row in the rowset that indicates whether a call to SQLSetPos() for a bulk operation is ignored or performed. Each element in the array is set to either SQL_ROW_PROCEED (the default) or SQL_ROW_IGNORE. This array is pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute.

The number of elements in the status and operation arrays must equal the number of rows in the rowset (as defined by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute).

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 146. SQLSetPos SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	Data truncated.	The *Operation* argument was SQL_REFRESH, and string or binary data returned for a column or columns with a data type of SQL_C_CHAR or SQL_C_BINARY resulted in the truncation of non-blank character or non-NULL binary data
01S01	Error in row.	The *RowNumber* argument was 0 and an error occurred in one or more rows while performing the operation specified with the *Operation* argument. (SQL_SUCCESS_WITH_INFO is returned if an error occurs on one or more, but not all, rows of a multirow operation, and SQL_ERROR is returned if an error occurs on a single-row operation.)

Table 146. SQLSetPos SQLSTATEs (continued)

SQLSTATE	Description	Explanation
01S07	Fractional truncation.	The *Operation* argument was SQL_REFRESH, the data type of the application buffer was not SQL_C_CHAR or SQL_C_BINARY, and the data returned to application buffers for one or more columns was truncated. For numeric data types, the fractional part of the number was truncated. For time and timestamp data types, the fractional portion of the time was truncated.
07006	Invalid conversion.	The data value of a column in the result set could not be converted to the data type specified by *TargetType* in the call to `SQLBindCol()`.
07009	Invalid descriptor index.	The argument *Operation* was SQL_REFRESH or SQL_UPDATE and a column was bound with a column number greater than the number of columns in the result set or a column number less than 0.
21S02	Degrees of derived table does not match column list.	The argument *Operation* was SQL_UPDATE and no columns were updateable because all columns were either unbound, read-only, or the value in the bound length/indicator buffer was SQL_COLUMN_IGNORE.
22001	String data right truncation.	The assignment of a character or binary value to a column resulted in the truncation of non-blank (for characters) or non-null (for binary) characters or bytes.
22003	Numeric value out of range.	The argument *Operation* was SQL_UPDATE and the assignment of a numeric value to a column in the result set caused the whole (as opposed to fractional) part of the number to be truncated. The argument *Operation* was SQL_REFRESH, and returning the numeric value for one or more bound columns would have caused a loss of significant digits.
22007	Invalid datetime format.	The argument *Operation* was SQL_UPDATE, and the assignment of a date or timestamp value to a column in the result set caused the year, month, or day field to be out of range. The argument *Operation* was SQL_REFRESH, and returning the date or timestamp value for one or more bound columns would have caused the year, month, or day field to be out of range.
22008	Datetime field overflow.	The *Operation* argument was SQL_UPDATE, and the performance of datetime arithmetic on data being sent to a column in the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result being outside the permissible range of values for the field, or being invalid based on the natural rules for datetimes based on the Gregorian calendar. The *Operation* argument was SQL_REFRESH, and the performance of datetime arithmetic on data being retrieved from the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result being outside the permissible range of values for the field, or being invalid based on the natural rules for datetimes based on the Gregorian calendar.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.

Table 146. SQLSetPos SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY010	Function sequence error.	The specified *StatementHandle* was not in an executed state. The function was called without first calling `SQLExecDirect()`, `SQLExecute()`, or a catalog function. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. `SQLExecute()`, `SQLExecDirect()`, or `SQLSetPos()` was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. An ODBC 2.0 application called SQLSetPos() for a *StatementHandle* before `SQLFetchScroll()` was called or after `SQLFetch()` was called, and before `SQLFreeStmt()` was called with the SQL_CLOSE option.
HY011	Operation invalid at this time.	An ODBC 2.0 application set the SQL_ATTR_ROW_STATUS_PTR statement attribute; then SQLSetPos() was called before `SQLFetch()`, `SQLFetchScroll()`, or `SQLExtendedFetch()` was called.
HY090	Invalid string or buffer length.	The *Operation* argument was SQL_ADD, SQL_UPDATE, or SQL_UPDATE_BY_BOOKMARK, a data value was a null pointer, and the column length value was not 0, SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. The *Operation* argument was SQL_ADD, SQL_UPDATE, or SQL_UPDATE_BY_BOOKMARK, a data value was not a null pointer, and the column length value was less than 0, but not equal to SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NTS, or SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. A value in a length/indicator buffer was SQL_DATA_AT_EXEC; the SQL type was either SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a other, data-source-specific data type; and the SQL_NEED_LONG_DATA_LEN information type in `SQLGetInfo()` was Y.
HY092	Option type out of range.	The *Operation* argument was SQL_UPDATE_BY_BOOKMARK, SQL_DELETE_BY_BOOKMARK, or SQL_REFRESH_BY_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.
HY107	Row value out of range.	The value specified for the argument *RowNumber* was greater than the number of rows in the rowset.

Table 146. SQLSetPos SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY109	Invalid cursor position.	The cursor associated with the *StatementHandle* was defined as forward only, so the cursor could not be positioned within the rowset. See the description for the SQL_ATTR_CURSOR_TYPE attribute in `SQLSetStmtAttr()`. The *Operation* argument was SQL_UPDATE, SQL_DELETE, or SQL_REFRESH, and the row identified by the *RowNumber* argument had been deleted or had not be fetched. The *RowNumber* argument was 0 and the *Operation* argument was SQL_POSITION.
HYC00	Driver not capable.	CLI or the data source does not support the operation requested in the *Operation* argument or the *LockType* argument.
HYT00	Timeout expired	The query timeout period expired before the data source returned the result set. The timeout period is set through `SQLSetStmtAttr()` with an *Attribute* of SQL_ATTR_QUERY_TIMEOUT.

Restrictions

None.

Example

```
/* set the cursor position in a rowset */
cliRC = SQLSetPos(hstmt, 3, SQL_POSITION, SQL_LOCK_NO_CHANGE);
```

SQLSetStmtAttr function (CLI) - Set options related to a statement

Sets options related to a statement.

To set an option for all statements associated with a specific connection, an application can call SQLSetConnectAttr().

Specification:

- **CLI** 5.0
- **ODBC** 3.0
- **ISO CLI**

Refer to the CLI statement attributes list for all available statement attributes.

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLSetStmtAttrW(). See "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLSetStmtAttr (
               SQLHSTMT         StatementHandle,  /* hstmt */
               SQLINTEGER       Attribute,        /* fOption */
               SQLPOINTER       ValuePtr,         /* pvParam */
               SQLINTEGER       StringLength);    /* fStrLen */
```

Function arguments

Table 147. SQLSetStmtAttr arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	input	Statement handle.
SQLINTEGER	*Attribute*	input	Option to set, described in the CLI statement attributes list.
SQLPOINTER	*ValuePtr*	input	Pointer to the value to be associated with *Attribute*. If *Attribute* is an ODBC-defined attribute, the application might need to qualify the attribute value in *ValuePtr* by setting the *StringLength* attribute as described in the *StringLength* description. If *Attribute* is a CLI attribute, the application should always qualify the attribute value in *ValuePtr* by setting the *StringLength* attribute as described in the *StringLength* description. **Note:** If *Attribute* is an ODBC attribute, *ValuePtr* can, depending on the attribute, be set to an unsigned integer. If *Attribute* is a CLI attribute, *ValuePtr* can, depending on the attribute, be set to a signed integer. If *ValuePtr* is set to a signed negative integer and an unsigned integer is expected, *ValuePtr* might be treated as a large unsigned integer by CLI without warning. Alternatively, CLI might return an error (SQLSTATE HY024).

Table 147. SQLSetStmtAttr arguments (continued)

Data type	Argument	Use	Description
SQLINTEGER	*StringLength*	input	If *Attribute* is an ODBC attribute, the application might need to qualify the attribute by setting *StringLength* to the following values: • If *ValuePtr* points to a character string or a binary buffer, *StringLength* should be the length of **ValuePtr*. For character string data, *StringLength* should contain the number of bytes in the string. • If *ValuePtr* is a pointer, but not to a string or binary buffer, then *StringLength* should have the value SQL_IS_POINTER. • If *ValuePtr* points to an unsigned integer, the *StringLength* attribute is ignored. If *Attribute* is a CLI attribute, the application must qualify the attribute by setting *StringLength* to the following values: • If *ValuePtr* is a pointer to a character string, then *StringLength* is the number of bytes needed to store the string or SQL_NTS. • If *ValuePtr* is a pointer to a binary buffer, then the application should place the result of the SQL_LEN_BINARY_ATTR (length) macro in *StringLength*. This places a negative value in *StringLength*. • If *ValuePtr* contains a fixed-length value, then *StringLength* is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate. • If *ValuePtr* is a pointer to a value other than a character string, a binary string, or a fixed-length value, then *StringLength* should have the value SQL_IS_POINTER.

Usage

Statement attributes for a statement remain in effect until they are changed by another call to SQLSetStmtAttr() or the statement is dropped by calling SQLFreeHandle(). Calling SQLFreeStmt() with the SQL_CLOSE, SQL_UNBIND, or SQL_RESET_PARAMS options does not reset statement attributes.

Some statement attributes support substitution of a similar value if the data source does not support the value specified in **ValuePtr*. In such cases, CLI returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed). For example, CLI supports a pure keyset cursor. As a result, CLI does not allow applications to change the default value of the SQL_ATTR_KEYSET_SIZE attribute. Instead, CLI substitutes SQL_KEYSET_SIZE_DEFAULT for all other values that might be supplied in the **ValuePtr* argument and returns SQL_SUCCESS_WITH_INFO. To determine the substituted value, an application calls SQLGetStmtAttr().

The format of information set with *ValuePtr* depends on the specified *Attribute*. SQLSetStmtAttr() accepts attribute information in one of two different formats: a null-terminated character string or a 32-bit integer value. The format of information returned in `SQLGetStmtAttr()` reflects what was specified in

SQLSetStmtAttr(). For example, character strings pointed to by the *ValuePtr* argument of SQLSetStmtAttr() have a length of *StringLength*, and this is the value that would be returned by SQLGetStmtAttr().

Setting statement attributes by setting descriptors

Many statement attributes also corresponding to a header field of one or more descriptors. These attributes can be set not only by a call to SQLSetStmtAttr(), but also by a call to SQLSetDescField(). Setting these options by a call to SQLSetStmtAttr(), rather than SQLSetDescField(), has the advantage that a descriptor handle does not have to be fetched.

Note: Calling SQLSetStmtAttr() for one statement can affect other statements. This occurs when the application parameter descriptor (APD) or application row descriptor (ARD) associated with the statement is explicitly allocated and is also associated with other statements. Because SQLSetStmtAttr() modifies the APD or ARD, the modifications apply to all statements with which this descriptor is associated. If this is not the required behavior, the application should dissociate this descriptor from the other statement (by calling SQLSetStmtAttr() to set the SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC field to a different descriptor handle) before calling SQLSetStmtAttr() again.

When a statement attribute that is also a descriptor field is set by a call to SQLSetStmtAttr(), the corresponding field in the descriptor that is associated with the statement is also set. The field is set only for the applicable descriptors that are currently associated with the statement identified by the *StatementHandle* argument, and the attribute setting does not affect any descriptors that might be associated with that statement in the future. When a descriptor field that is also a statement attribute is set by a call to SQLSetDescField(), the corresponding statement attribute is also set.

Statement attributes determine which descriptors a statement handle is associated with. When a statement is allocated (see SQLAllocHandle()), four descriptor handles are automatically allocated and associated with the statement. Explicitly allocated descriptor handles can be associated with the statement by calling SQLAllocHandle() with a *HandleType* of SQL_HANDLE_DESC to allocate a descriptor handle, then calling SQLSetStmtAttr() to associate the descriptor handle with the statement.

The following statement attributes correspond to descriptor header fields:

Table 148. Statement attributes

Statement attribute	Header field	Descriptor
SQL_ATTR_PARAM_BIND_OFFSET_PTR	SQL_DESC_BIND_OFFSET_PTR	APD
SQL_ATTR_PARAM_BIND_TYPE	SQL_DESC_BIND_TYPE	APD
SQL_ATTR_PARAM_OPERATION_PTR	SQL_DESC_ARRAY_STATUS_PTR	APD
SQL_ATTR_PARAM_STATUS_PTR	SQL_DESC_ARRAY_STATUS_PTR	IPD
SQL_ATTR_PARAMS_PROCESSED_PTR	SQL_DESC_ROWS_PROCESSED_PTR	IPD
SQL_ATTR_PARAMSET_SIZE	SQL_DESC_ARRAY_SIZE	APD
SQL_ATTR_ROW_ARRAY_SIZE	SQL_DESC_ARRAY_SIZE	APD
SQL_ATTR_ROW_BIND_OFFSET_PTR	SQL_DESC_BIND_OFFSET_PTR	ARD
SQL_ATTR_ROW_BIND_TYPE	SQL_DESC_BIND_TYPE	ARD
SQL_ATTR_ROW_OPERATION_PTR	SQL_DESC_ARRAY_STATUS_PTR	APD
SQL_ATTR_ROW_STATUS_PTR	SQL_DESC_ARRAY_STATUS_PTR	IRD
SQL_ATTR_ROWS_FETCHED_PTR	SQL_DESC_ROWS_PROCESSED_PTR	IRD

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 149. SQLSetStmtAttr SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed.	CLI did not support the value specified in **ValuePtr*, or the value specified in **ValuePtr* was invalid because of SQL constraints or requirements, so CLI substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication link failure.	The communication link between CLI and the data source to which it was connected failed before the function completed processing.
24000	Invalid cursor state.	The *Attribute* was SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE, SQL_ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS and the cursor was open.
HY000	General error.	An error occurred for which there was no specific SQLSTATE. The error message returned by `SQLGetDiagRec()` in the **MessageText* buffer describes the error and its cause.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY009	Invalid argument value.	A null pointer was passed for *ValuePtr* and the value in **ValuePtr* was a string attribute.
HY010	Function sequence error.	An asynchronously executing function was called for the *StatementHandle* and was still executing when this function was called. SQLExecute() or SQLExecDirect() was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY011	Operation invalid at this time.	The *Attribute* was SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE, SQL_ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS and the statement was prepared.
HY017	Invalid use of an automatically allocated descriptor handle.	The *Attribute* argument was SQL_ATTR_IMP_ROW_DESC or SQL_ATTR_IMP_PARAM_DESC. The *Attribute* argument was SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and the value in **ValuePtr* was an implicitly allocated descriptor handle.
HY024	Invalid attribute value.	Given the specified *Attribute* value, an invalid value was specified in **ValuePtr*. (CLI returns this SQLSTATE only for connection and statement attributes that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE.)
HY090	Invalid string or buffer length.	The *StringLength* argument was less than 0, but was not SQL_NTS.

Table 149. SQLSetStmtAttr SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY092	Option type out of range.	The value specified for the argument *Attribute* was not valid for this version of CLI.
HYC00	Driver not capable.	The value specified for the argument *Attribute* was a valid connection or statement attribute for the version of the CLI driver, but was not supported by the data source.

Restrictions

None.

Example

```
/* set the required statement attributes */
cliRC = SQLSetStmtAttr(hstmt,
                       SQL_ATTR_ROW_ARRAY_SIZE,
                       (SQLPOINTER)ROWSET_SIZE,
                       0);
STMT_HANDLE_CHECK(hstmt, hdbc, cliRC);

/* set the required statement attributes */
cliRC = SQLSetStmtAttr(hstmt,
                       SQL_ATTR_ROW_BIND_TYPE,
                       SQL_BIND_BY_COLUMN,
                       0);
STMT_HANDLE_CHECK(hstmt, hdbc, cliRC);

/* set the required statement attributes */
cliRC = SQLSetStmtAttr(hstmt,
                       SQL_ATTR_ROWS_FETCHED_PTR,
                       &rowsFetchedNb,
                       0);
STMT_HANDLE_CHECK(hstmt, hdbc, cliRC);
```

SQLSetStmtOption function (CLI) - Set statement option

In ODBC 3.0, SQLSetStmtOption() has been deprecated and replaced with SQLSetStmtAttr().

Although this version of CLI continues to support SQLSetStmtOption(), use SQLSetStmtAttr() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

Note: This deprecated function cannot be used in a 64-bit environment.

The statement:

```
SQLSetStmtOption(
           hstmt,
           SQL_ROWSET_SIZE,
           RowSetSize);
```

for example, would be rewritten using the new function as:

```
SQLSetStmtAttr(
          hstmt,
          SQL_ATTR_ROW_ARRAY_SIZE,
          (SQLPOINTER) RowSetSize,
          0);
```

SQLSpecialColumns function (CLI) - Get special (row identifier) columns

Returns unique row identifier information (for example, the primary key or unique index) for a table.

The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Specification:

- **DB2 Call Level Interface** 2.1
- **ODBC** 1.0

Unicode equivalent: This function can also be used with the Unicode character set. The corresponding Unicode function is SQLSpecialColumnsW(). Refer to "Unicode functions (CLI)" on page 5 for information about ANSI to Unicode function mappings.

Syntax

```
SQLRETURN   SQLSpecialColumns(
               SQLHSTMT          StatementHandle,   /* hstmt */
               SQLUSMALLINT      IdentifierType,    /* fColType */
               SQLCHAR           *CatalogName,      /* szCatalogName */
               SQLSMALLINT       NameLength1,       /* cbCatalogName */
               SQLCHAR           *SchemaName,       /* szSchemaName */
               SQLSMALLINT       NameLength2,       /* cbSchemaName */
               SQLCHAR           *TableName,        /* szTableName */
               SQLSMALLINT       NameLength3,       /* cbTableName */
               SQLUSMALLINT      Scope,             /* fScope */
               SQLUSMALLINT      Nullable);         /* fNullable */
```

Function arguments

Table 150. SQLSpecialColumns arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	Statement handle
SQLUSMALLINT	*IdentifierType*	Input	Type of unique row identifier to return. Only the listed types are supported: • SQL_BEST_ROWID Returns the optimal set of column(s) which can uniquely identify any row in the specified table. **Note:** For compatibility with ODBC applications, SQL_ROWVER is also recognized, but not supported; therefore, if SQL_ROWVER is specified, an empty result will be returned.

Table 150. SQLSpecialColumns arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*CatalogName*	Input	Catalog qualifier of a 3-part table name. If the target DBMS does not support 3-part naming, and *CatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS will be returned. Otherwise, this is a valid filter for DBMSs that support 3-part naming.
SQLSMALLINT	*NameLength1*	Input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.
SQLCHAR *	*SchemaName*	Input	Schema qualifier of the specified table.
SQLSMALLINT	*NameLength2*	Input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*TableName*	Input	Table name.
SQLSMALLINT	*NameLength3*	Input	Number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) needed to store *TableName*, or SQL_NTS if *TableName* is null-terminated.
SQLUSMALLINT	*Scope*	Input	Minimum required duration for which the unique row identifier will be valid. *Scope* must be one of the SQL_SCOPE_CURROW, SQL_SCOPE_TRANSACTION, or SQL_SCOPE_SESSION: • SQL_SCOPE_CURROW: The row identifier is guaranteed to be valid only while positioned on that row. A later re-select using the same row identifier values might not return a row if the row was updated or deleted by another transaction. • SQL_SCOPE_TRANSACTION: The row identifier is guaranteed to be valid for the duration of the current transaction. • SQL_SCOPE_SESSION: The row identifier is guaranteed to be valid for the duration of the connection. The duration over which a row identifier value is guaranteed to be valid depends on the current transaction isolation level.
SQLUSMALLINT	*Nullable*	Input	Determines whether to return special columns that can have a NULL value. Must be SQL_NO_NULL or SQL_NULLABLE: • SQL_NO_NULLS - The row identifier column set returned cannot have any NULL values. • SQL_NULLABLE - The row identifier column set returned might include columns where NULL values are permitted.

Usage

If multiple ways exist to uniquely identify any row in a table (for example, if there are multiple unique indexes on the specified table), then DB2 Call Level Interface will return the *best* set of row identifier column set based on its internal criterion.

If the schema qualifier argument associated with a table name is not specified, then the schema name defaults to the one currently in effect for the current connection.

If there is no column set which allows any row in the table to be uniquely identified, an empty result set is returned.

The unique row identifier information is returned in the form of a result set where each column of the row identifier is represented by one row in the result set. Columns returned by SQLSpecialColumns shows the order of the columns in the result set returned by SQLSpecialColumns(), sorted by SCOPE.

Since calls to SQLSpecialColumns() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

Call SQLGetInfo() with the SQL_MAX_COLUMN_NAME_LEN to determine the actual length of the COLUMN_NAME column supported by the connected DBMS.

Although new columns might be added and the names of the columns changed in future releases, the position of the current columns will not change.

Note: IDS data servers have a virtual column named ROWID for every non-fragmented table. The SQLSpecialColumns() function will return information about the ROWID column when accessing IDS data servers.

Columns returned by SQLSpecialColumns

Column 1 SCOPE (SMALLINT)
The duration for which the name in COLUMN_NAME is guaranteed to point to the same row. Valid values are the same as for the *Scope* argument: Actual scope of the row identifier. Contains one of the listed values:
- SQL_SCOPE_CURROW
- SQL_SCOPE_TRANSACTION
- SQL_SCOPE_SESSION

Refer to *Scope* in Table 150 on page 295 for a description of each value.

Column 2 COLUMN_NAME (VARCHAR(128) not NULL)
Name of the column that is (or is part of) the table's primary key.

Column 3 DATA_TYPE (SMALLINT not NULL)
SQL data type of the column.

Column 4 TYPE_NAME (VARCHAR(128) not NULL)
DBMS character string representation of the name associated with DATA_TYPE column value.

Column 5 COLUMN_SIZE (INTEGER)
If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in bytes; if it is a graphic (DBCS) string, this is the number of double byte characters for the parameter.

For date, time, timestamp data types, this is the total number of SQLCHAR or SQLWCHAR elements required to display the value when converted to character.

For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.

Refer to the table of data type precision.

Column 6 BUFFER_LENGTH (INTEGER)
The maximum number of bytes for the associated C buffer to store data from this column if SQL_C_DEFAULT were specified on the SQLBindCol(), SQLGetData() and SQLBindParameter() calls. This length does not include any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign.

Refer to the table of data type length.

Column 7 DECIMAL_DIGITS (SMALLINT)
The scale of the column. NULL is returned for data types where scale is not applicable. Refer to the table of data type scale.

Column 8 PSEUDO_COLUMN (SMALLINT)
Indicates whether or not the column is a pseudo-column DB2 Call Level Interface will only return:

- SQL_PC_NOT_PSEUDO

DB2 DBMSs do not support pseudo-columns. ODBC applications might receive the listed values from RDBMS servers that are not from IBM:
- SQL_PC_UNKNOWN
- SQL_PC_PSEUDO

Return codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 151. SQLSpecialColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY009	Invalid argument value.	*TableName* is null.

Table 151. SQLSpecialColumns SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the length arguments was less than 0, but not equal to SQL_NTS. The value of one of the length arguments exceeded the maximum length supported by the DBMS for that qualifier or name.
HY097	Column type out of range.	An invalid *IdentifierType* value was specified.
HY098	Scope type out of range.	An invalid *Scope* value was specified.
HY099	Nullable type out of range.	An invalid *Nullable* values was specified.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

Example

```
/* get special columns */
cliRC = SQLSpecialColumns(hstmt,
                          SQL_BEST_ROWID,
                          NULL,
                          0,
                          tbSchema,
                          SQL_NTS,
                          tbName,
                          SQL_NTS,
                          SQL_SCOPE_CURROW,
                          SQL_NULLABLE);
```

SQLStatistics function (CLI) - Get index and statistics information for a base table

The `SQLStatistics()` function retrieves index information for a given table.

The `SQLStatistics()` function also returns the cardinality and the number of pages that are associated with the table and the indexes on the table.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

The information is returned in a result set, which you can retrieve by using the same functions that you use to process a result set that is generated by a query.

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is `SQLStatisticsW()`. For information about ANSI to Unicode function mappings, see "Unicode functions (CLI)" on page 5.

Syntax

```
SQLRETURN   SQLStatistics    (
               SQLHSTMT          StatementHandle,  /* hstmt */
               SQLCHAR           *CatalogName,     /* szCatalogName */
               SQLSMALLINT       NameLength1,      /* cbCatalogName */
               SQLCHAR           *SchemaName,      /* szSchemaName */
               SQLSMALLINT       NameLength2,      /* cbSchemaName */
               SQLCHAR           *TableName,       /* szTableName */
               SQLSMALLINT       NameLength3,      /* cbTableName */
               SQLUSMALLINT      Unique,           /* fUnique */
               SQLUSMALLINT      Reserved);        /* fAccuracy */
```

Function arguments

Table 152. SQLStatistics arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	The statement handle.
SQLCHAR *	*CatalogName*	Input	A catalog qualifier of a 3-part table name. If the target DBMS does not support 3-part naming, and *CatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS isreturned. Otherwise, this is a valid filter for DBMSs that supports 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.
SQLCHAR *	*SchemaName*	Input	The schema qualifier of the specified table.
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*TableName*	Input	The table name.
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *TableName*, or SQL_NTS if *TableName* is null-terminated.
SQLUSMALLINT	*Unique*	Input	The type of index information to return: • SQL_INDEX_UNIQUE Only unique indexes are returned. • SQL_INDEX_ALL All indexes are returned.

Table 152. SQLStatistics arguments (continued)

Data type	Argument	Use	Description
SQLUSMALLINT	*Reserved*	Input	Indicates whether the CARDINALITY and PAGES columns in the result set contain the most current information: • SQL_ENSURE : This value is reserved for future use, when the application requests the most up-to-date statistics information. **New applications should not use this value**. Existing applications that specify this value will receive the same results as SQL_QUICK. • SQL_QUICK : Statistics which are readily available at the server are returned. No attempt is made to ensure that the values are current.

Usage

The `SQLStatistics()` function returns two types of information:

- Statistics information for the table (if it is available):
  - if the TYPE column of the result set is set to SQL_TABLE_STAT, the number of rows in the table, and the number of pages that are used to store the table are returned.
  - if the TYPE column of the result set indicates an index, the number of unique values in the index, and the number of pages that are used to store the indexes are returned.
- Information about each index, where each index column is represented by one row of the result set. The result set columns are described in Columns returned by SQLStatistics. The rows in the result set are ordered by NON_UNIQUE, TYPE, INDEX_QUALIFIER, INDEX_NAME and KEY_SEQ columns.

In many cases, calls to the `SQLStatistics()` function map to a complex and thus expensive query against the system catalog, so you should use the calls sparingly, and save the results rather than repeating calls.

If the schema qualifier argument that is associated with a table name is not specified, the schema name defaults to the argument that is in effect for the current connection.

Call the `SQLGetInfo()` function with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns that are supported by the connected DBMS.

You can specify *ALL as a value in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. CLI searches on all existing schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns returned by SQLStatistics

Column 1 TABLE_CAT (VARCHAR(128))
The catalog name of the table for which the index applies. The value is NULL if this table does not have catalogs.

Column 2 TABLE_SCHEM (VARCHAR(128))
The name of the schema containing TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128) not NULL)
The name of the table.

Column 4 NON_UNIQUE (SMALLINT)
Indicates whether the index prohibits duplicate values:
- SQL_TRUE is returned if the index allows duplicate values.
- SQL_FALSE is returned if the index values must be unique.
- NULL is returned if the TYPE column indicates that this row is SQL_TABLE_STAT (statistics information about the table).

Column 5 INDEX_QUALIFIER (VARCHAR(128))
The string that is used to qualify the index name in the DROP INDEX statement. Appending a period (.) plus the INDEX_NAME results in a full specification of the index.

Column 6 INDEX_NAME (VARCHAR(128))
The name of the index. If the TYPE column has the value SQL_TABLE_STAT, this column has the value NULL.

Column 7 TYPE (SMALLINT not NULL)
Indicates the type of information that is contained in this row of the result set:
- SQL_TABLE_STAT indicates that this row contains statistics information about the table.
- SQL_INDEX_CLUSTERED indicates that this row contains information about an index, and the index type is a clustered index.
- SQL_INDEX_HASHED indicates that this row contains information about an index, and the index type is a hashed index.
- SQL_INDEX_OTHER indicates that this row contains information about an index that is not clustered or hashed

Column 8 ORDINAL_POSITION (SMALLINT)
The ordinal position of the column in the index whose name is given in the INDEX_NAME column. A NULL value is returned for this column if the TYPE column has the value of SQL_TABLE_STAT.

Column 9 COLUMN_NAME (VARCHAR(128))
The name of the column in the index. A NULL value is returned for this column if the TYPE column has the value of SQL_TABLE_STAT.

Column 10 ASC_OR_DESC (CHAR(1))
The sort sequence for the column; "**A**" for ascending, or "**D**" for descending. A NULL value is returned if the value in the TYPE column is SQL_TABLE_STAT.

Column 11 CARDINALITY (INTEGER)
- If the TYPE column contains the value SQL_TABLE_STAT, this column contains the number of rows that are in the table.
- If the TYPE column value is not SQL_TABLE_STAT, this column contains the number of unique values that are in the index.
- A NULL value is returned if information is not available from the DBMS.

Column 12 PAGES (INTEGER)

- If the TYPE column contains the value SQL_TABLE_STAT, this column contains the number of pages that are used to store the table.
- If the TYPE column value is not SQL_TABLE_STAT, this column contains the number of pages that are used to store the indexes.
- A NULL value is returned if information is not available from the DBMS.

Column 13 FILTER_CONDITION (VARCHAR(128))

If the index is a filtered index, this is the filter condition. Because DB2 servers do not support filtered indexes, NULL is always returned. NULL is also returned if TYPE is SQL_TABLE_STAT.

For the row in the result set that contains table statistics (TYPE is set to SQL_TABLE_STAT), the columns values of NON_UNIQUE, INDEX_QUALIFIER, INDEX_NAME, ORDINAL_POSITION, COLUMN_NAME, and ASC_OR_DESC are set to NULL. If the CARDINALITY or PAGES information cannot be determined, then NULL is returned for those columns.

Note: An application can check the SQLERRD(3) and SQLERRD(4) fields of the SQLCA to gather some statistics on a table. However, the accuracy of the information that is returned in those fields depends on many factors, such as the use of parameter markers and expressions within the statement. The main factor that you can control is the accuracy of the database statistics. For example, for DB2 Database for Linux, UNIX, and Windows, the last time the `RUNSTATS` command was run. Therefore, the statistics information that is returned by `SQLStatistics()` is often more consistent and reliable than the statistics information that is contained in the SQLCA fields that were previously explained.

Return codes

- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_STILL_EXECUTING
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

Diagnostics

Table 153. SQLStatistics SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY009	Invalid argument value.	*TableName* is null.

Table 153. SQLStatistics SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while within a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS. The valid of one of the name-length arguments exceeded the maximum value that is supported for that data source. The maximum supported value can be obtained by calling the `SQLGetInfo()` function.
HY100	Uniqueness option type out of range.	An invalid *Unique* value was specified.
HY101	Accuracy option type out of range.	An invalid *Reserved* value was specified.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. You can set the timeout period by using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

Example

```
/* get index and statistics information for a base table */
cliRC = SQLStatistics(hstmt,
                      NULL,
                      0,
                      tbSchema,
                      SQL_NTS,
                      tbName,
                      SQL_NTS,
                      SQL_INDEX_UNIQUE,
                      SQL_QUICK);
```

SQLTablePrivileges function (CLI) - Get privileges associated with a table

The `SQLTablePrivileges()` function returns a list of tables and associated privileges for each table.

The information is returned in an SQL result set, which you can retrieve by using the same functions that you use to process a result set that is generated by a query.

SQLTablePrivileges function (CLI) - Get privileges associated with a table

Specification:

- **CLI** 2.1
- **ODBC** 1.0

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is `SQLTablePrivilegesW()`. For details about ANSI to Unicode function mappings, see"Unicode functions (CLI)" on page 5.

Syntax

```
SQLRETURN SQLTablePrivileges (
            SQLHSTMT          StatementHandle,   /* hstmt */
            SQLCHAR           *CatalogName,      /* *szCatalogName */
            SQLSMALLINT       NameLength1,       /* cbCatalogName */
            SQLCHAR           *SchemaName,       /* *szSchemaName */
            SQLSMALLINT       NameLength2,       /* cbSchemaName */
            SQLCHAR           *TableName,        /* *szTableName */
            SQLSMALLINT       NameLength3);      /* cbTableName */
```

Function arguments

Table 154. SQLTablePrivileges arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	The statement handle.
SQLCHAR *	*CatalogName*	Input	The catalog qualifier of a 3-part table name. If the target DBMS does not support 3-part naming, and *PKCatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS is returned. Otherwise, this is a valid filter for DBMSs that supports 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.
SQLCHAR *	*SchemaName*	Input	A buffer that can contain a *pattern value* to qualify the result set by schema name.
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*TableName*	Input	A buffer that can contain a *pattern value* to qualify the result set by table name.
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *TableName*, or SQL_NTS if *TableName* is null-terminated.

Note that the *SchemaName* and *TableName* input arguments accept search patterns.

Usage

The results are returned as a standard result set that contain the columns that are listed in the succeeding table. The result set is ordered by the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and PRIVILEGE columns. If multiple privileges are associated with any given table, each privilege is returned as a separate row.

The granularity of each privilege that is reported here might apply at the column level. For example, for some data sources, if you can update a table, you can also update every column in that table. For other data sources, the application must call `SQLColumnPrivileges()` to discover if the individual columns have the same table privileges.

In many cases, calls to the `SQLTablePrivileges()` function map to a complex and thus expensive query against the system catalog, so you should use the calls sparingly, and save the results rather than repeating the calls.

Sometimes, an application calls the function and no attempt is made to restrict the result set that is returned. For some data sources that contain a large quantity of tables, views, and aliases for example, this scenario maps to an extremely large result set and very long retrieval times. In order to help reduce the long retrieval times, you can specify the SchemaList configuration keyword in the CLI initialization file to help restrict the result set when the application has supplied a null pointer for the SchemaName. If the application specifies a SchemaName string, the SchemaList keyword is still used to restrict the output. Therefore, if the schema name that is supplied is not in the SchemaList string, the result is an empty result set.

Call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns that are supported by the connected DBMS.

You can specify *ALL or *USRLIBL as values in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. If you specify *ALL, CLI searches on all existing schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. For IBM DB2 for IBM i servers, if you specify *USRLIBL, CLI searches on the current libraries of the server job. For other DB2 servers, *USRLIBL does not have a special meaning and CLI searches using *USRLIBL as a pattern. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL or *USRLIBL.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns returned by SQLTablePrivileges

Column 1 TABLE_CAT (VARCHAR(128))
The name of the catalog table. The value is NULL if this table does not have catalogs.

Column 2 TABLE_SCHEM (VARCHAR(128))
The name of the schema that contains TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128) not NULL)
The name of the table.

Column 4 GRANTOR (VARCHAR(128))
The authorization ID of the user who granted the privilege.

Column 5 GRANTEE (VARCHAR(128))
The authorization ID of the user to whom the privilege is granted.

Column 6 PRIVILEGE (VARCHAR(128))
The table privilege. This privilege can be one of the listed strings:
- ALTER
- CONTROL
- DELETE
- INDEX
- INSERT
- REFERENCES
- SELECT
- UPDATE

Column 7 IS_GRANTABLE (VARCHAR(3))
Indicates whether the grantee is permitted to grant the privilege to other users.

The IS_GRANTABLE value can be "YES", "NO," or NULL.

Note: The column names that are used by CLI follow the X/Open CLI CAE specification style. The column types, contents, and order are identical to those that are defined for the `SQLProcedures()` result set in ODBC.

Return codes

- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_STILL_EXECUTING
- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

Diagnostics

Table 155. SQLTablePrivileges SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.

Table 155. SQLTablePrivileges SQLSTATEs (continued)

SQLSTATE	Description	Explanation
HY010	Function sequence error.	The function was called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called during a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name-length arguments was less than 0, but not equal to SQL_NTS. The valid value of one of the name-length arguments exceeded the maximum value that is supported for that data source. You can obtain the maximum supported value by calling the `SQLGetInfo()` function.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. You can set the timeout period by using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

Example

```
/* get privileges associated with a table */
cliRC = SQLTablePrivileges(hstmt,
                           NULL,
                           0,
                           tbSchemaPattern,
                           SQL_NTS,
                           tbNamePattern,
                           SQL_NTS);
```

SQLTables function (CLI) - Get table information

The `SQLTables()` function returns a list of table names and associated information that is stored in the system catalog of the connected data source.

The list of table names is returned as a result set, which you can retrieve by using the same functions that you use to process a result set that is generated by a query.

Specification:

- **CLI** 2.1
- **ODBC** 1.0

Unicode equivalent: You can also use this function with the Unicode character set. The corresponding Unicode function is `SQLTablesW()`. For information about ANSI and Unicode function mappings, see "Unicode functions (CLI)" on page 5.

Syntax

```
SQLRETURN   SQLTables       (
               SQLHSTMT         StatementHandle,   /* hstmt */
               SQLCHAR          *CatalogName,      /* szCatalogName */
               SQLSMALLINT      NameLength1,       /* cbCatalogName */
               SQLCHAR          *SchemaName,       /* szSchemaName */
               SQLSMALLINT      NameLength2,       /* cbSchemaName */
               SQLCHAR          *TableName,        /* szTableName */
               SQLSMALLINT      NameLength3,       /* cbTableName */
               SQLCHAR          *TableType,        /* szTableType */
               SQLSMALLINT      NameLength4);      /* cbTableType */
```

Function arguments

Table 156. SQLTables arguments

Data type	Argument	Use	Description
SQLHSTMT	*StatementHandle*	Input	The statement handle.
SQLCHAR *	*CatalogName*	Input	A catalog qualifier of a 3-part table name that can contain a *pattern value*. If the target DBMS does not support 3-part naming, and *CatalogName* is not a null pointer and does not point to a zero-length string, then an empty result set and SQL_SUCCESS is returned. Otherwise, this is a valid filter for DBMSs that support 3-part naming.
SQLSMALLINT	*NameLength1*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *CatalogName*, or SQL_NTS if *CatalogName* is null-terminated.
SQLCHAR *	*SchemaName*	Input	A buffer that can contain a *pattern value* to qualify the result set by the schema name.
SQLSMALLINT	*NameLength2*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *SchemaName*, or SQL_NTS if *SchemaName* is null-terminated.
SQLCHAR *	*TableName*	Input	A buffer that can contain a *pattern value* to qualify the result set by the table name.
SQLSMALLINT	*NameLength3*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *TableName*, or SQL_NTS if *TableName* is null-terminated.

Table 156. SQLTables arguments (continued)

Data type	Argument	Use	Description
SQLCHAR *	*TableType*	Input	A buffer that can contain a *value list* to qualify the result set by table type. The value list is a list of uppercase comma-separated single values for the table types of interest. Valid table type identifiers include: ALIAS, HIERARCHY TABLE, INOPERATIVE VIEW, NICKNAME, MATERIALIZED QUERY TABLE, SYSTEM TABLE, TABLE, TYPED TABLE, TYPED VIEW, or VIEW. If *TableType* argument is a NULL pointer or a zero length string, this is equivalent to specifying all of the possibilities for the table type identifier. If SYSTEM TABLE is specified, both system tables and system views (if there are any) are returned.
SQLSMALLINT	*NameLength4*	Input	The number of SQLCHAR elements (or SQLWCHAR elements for the Unicode variant of this function) that are required to store *TableType*, or SQL_NTS if *TableType* is null-terminated.

The *CatalogName*, *SchemaName*, and *TableName* input arguments accept search patterns.

Usage

Table information is returned in a result set where each table is represented by one row of the result set. To determine the type of access that is permitted on any given table in the list, the application can call the `SQLTablePrivileges()` function. The application must be able to handle a situation where the user selects a table for which SELECT privileges are not granted.

To support obtaining just a list of schemas, you can apply the succeeding semantics for the *SchemaName* argument: if *SchemaName* is a string that contains a single percent (%) character, and *CatalogName* and *TableName* are empty strings, then the result set contains a list of valid schemas in the data source.

If *TableType* is a single percent character (%) and *CatalogName, SchemaName*, and *TableName* are empty strings, then the result set contains a list of valid table types for the data source. (All columns except the TABLE_TYPE column contain NULLs.)

If *TableType* is not an empty string, it must contain a list of uppercase, comma-separated values for the types of interest. You can enclose each value in single quotation marks or place double quotation marks around all of the values. For example, "'TABLE','VIEW'" or "TABLE,VIEW". If the data source does not support or does not recognize a specified table type, nothing is returned for that type.

Sometimes, an application calls the `SQLTables()` function with null pointers for some or all of the *SchemaName, TableName*, and *TableType* arguments so that no attempt is made to restrict the result set that is returned. For some data sources that contain a large quantity of tables, views and, aliases for example, this scenario maps to an extremely large result set and very long retrieval times. You can specify three configuration keywords (SCHEMALIST, SYSSCHEMA, TABLETYPE) in the CLI initialization file to help restrict the result set when the application has

supplied null pointers for *SchemaName*, *TableType*, or both. If the application specifies a *SchemaName* string, the SCHEMALIST keyword is still used to restrict the output. Therefore, if the schema name that is supplied is not in the SCHEMALIST string, the result is an empty result set.

The result set that is returned by the `SQLTables()` function contains the columns that are listed in Columns returned by SQLTables in the order given. The rows are ordered by the TABLE_TYPE, TABLE_CAT, TABLE_SCHEM, and TABLE_NAME columns.

In many cases, calls to the `SQLTables()` function map to a complex and thus expensive query against the system catalog, so you should use the calls sparingly, and save the results rather than repeating calls.

Call the `SQLGetInfo()` function with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns that are supported by the connected DBMS.

You can specify *ALL or *USRLIBL as values in the *SchemaName* to resolve unqualified stored procedure calls or to find libraries in catalog API calls. If you specify *ALL, CLI searches on all existing schemas in the connected database. You are not required to specify *ALL, as this behavior is the default in CLI. For IBM DB2 for IBM i servers, if you specify *USRLIBL, CLI searches on the current libraries of the server job. For other DB2 servers, *USRLIBL does not have a special meaning and CLI searches using *USRLIBL as a pattern. Alternatively, you can set the SchemaFilter IBM Data Server Driver configuration keyword or the Schema List CLI/ODBC configuration keyword to *ALL or *USRLIBL.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns will not change.

Columns returned by SQLTables

Column 1 TABLE_CAT (VARCHAR(128))
Name of the catalog containing TABLE_SCHEM. The value is NULL if this table does not have catalogs.

Column 2 TABLE_SCHEM (VARCHAR(128))
Name of the schema containing TABLE_NAME.

Column 3 TABLE_NAME (VARCHAR(128))
Name of the table, view, alias, or synonym.

Column 4 TABLE_TYPE (VARCHAR(128))
Identifies the type that is given by the name in the TABLE_NAME column. It can have the string values 'ALIAS', 'HIERARCHY TABLE', 'INOPERATIVE VIEW', 'NICKNAME', 'MATERIALIZED QUERY TABLE', 'SYSTEM TABLE', 'TABLE', 'TYPED TABLE', 'TYPED VIEW', or 'VIEW'.

Column 5 REMARKS (VARCHAR(254))
Descriptive information about the table.

Return codes

- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_STILL_EXECUTING

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

Diagnostics

Table 157. SQLTables SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source failed before the function completed.
HY001	Memory allocation failure.	DB2 CLI is unable to allocate memory required to support execution or completion of the function. It is likely that process-level memory has been exhausted for the application process. Consult the operating system configuration for information about process-level memory limitations.
HY008	Operation was Canceled.	Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, `SQLCancel()` was called on *StatementHandle* from a different thread in a multithreaded application. Then the function was called again on *StatementHandle*.
HY009	Invalid argument value.	*TableName* is null.
HY010	Function sequence error.	The function was called during a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation. The function was called while in a BEGIN COMPOUND and END COMPOUND SQL operation. An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called. The function was called before a statement was prepared on the statement handle.
HY014	No more handles.	DB2 CLI was unable to allocate a handle due to resource limitations.
HY090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS. The valid of one of the name length arguments exceeded the maximum value supported for that data source. The maximum supported value can be obtained by calling the `SQLGetInfo()` function.
HYT00	Timeout expired.	The timeout period expired before the data source returned the result set. The timeout period can be set using the SQL_ATTR_QUERY_TIMEOUT attribute for `SQLSetStmtAttr()`.

Restrictions

None.

Example

```
/* get table information */
cliRC = SQLTables(hstmt,
                  NULL,
                  0,
```

```
tbSchemaPattern,
SQL_NTS,
tbNamePattern,
SQL_NTS,
NULL,
0);
```

SQLTransact function (CLI) - Transaction management

In ODBC 3.0, SQLTransact() has been deprecated and replaced with SQLEndTran().

Although this version of CLI continues to support SQLTransact(), use SQLEndTran() in your CLI programs so that they conform to the latest standards.

Migrating to the new function

The statement:

```
SQLTransact(henv, hdbc, SQL_COMMIT);
```

for example, would be rewritten using the new function as:

```
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
```

Chapter 2. Return codes and SQLSTATES for CLI

When you call CLI functions, there are two levels of diagnostics returned by the function: return codes and detailed diagnostics (SQLSTATEs, messages, SQLCA).

Each CLI function returns the function return code as a basic diagnostic. Both `SQLGetDiagRec()` and `SQLGetDiagField()` provide more detailed diagnostic information. If the diagnostic originates at the DBMS, the `SQLGetSQLCA()` function provides access to the SQLCA. This arrangement lets applications handle the basic flow control based on return codes, and use the SQLSTATES along with the SQLCA to determine the specific causes of failure and to perform specific error handling.

Both `SQLGetDiagRec()` and `SQLGetDiagField()` return three pieces of information:

- SQLSTATE
- Native error: if the diagnostic is detected by the data source, this is the SQLCODE; otherwise, this is set to -99999.
- Message text: this is the message text associated with the SQLSTATE.

`SQLGetSQLCA()` returns the SQLCA for access to specific fields, but should only be used when `SQLGetDiagRec()` or `SQLGetDiagField()` cannot provide the required information.

CLI function return codes

The following table lists all possible return codes for CLI functions.

Table 158. CLI Function return codes

Return code	Explanation
SQL_SUCCESS	The function completed successfully, no additional SQLSTATE information is available.
SQL_SUCCESS_WITH_INFO	The function completed successfully with a warning or other information. Call `SQLGetDiagRec()` or `SQLGetDiagField()` to receive the SQLSTATE and any other informational messages or warnings. The SQLSTATE will have a class of '01'.
SQL_STILL_EXECUTING	The function is running asynchronously and has not yet completed. The CLI driver has returned control to the application after calling the function, but the function has not yet finished executing.
SQL_NO_DATA_FOUND	The function returned successfully, but no relevant data was found. When this is returned after the execution of an SQL statement, additional information may be available and can be obtained by calling `SQLGetDiagRec()` or `SQLGetDiagField()`.
SQL_NEED_DATA	The application tried to execute an SQL statement but CLI lacks parameter data that the application had indicated would be passed at execute time.
SQL_ERROR	The function failed. Call `SQLGetDiagRec()` or `SQLGetDiagField()` to receive the SQLSTATE and any other error information.

Table 158. CLI Function return codes (continued)

Return code	Explanation
SQL_INVALID_HANDLE	The function failed due to an invalid input handle (environment, connection or statement handle). This is a programming error. No further information is available.

The following code segment shows how a function return code, SQL_NO_DATA_FOUND, can be used to control when data retrieval should stop:

```
while (cliRC != SQL_NO_DATA_FOUND)
{
  printf("     %-8d %-14.14s \n", deptnumb.val, location.val);
     /* fetch next row */
  cliRC = SQLFetch(hstmt);
  STMT_HANDLE_CHECK(hstmt, hdbc, cliRC);
}
```

SQLSTATES for CLI

SQLSTATEs are alphanumeric strings of 5 characters (bytes) with a format of `ccsss`, where `cc` indicates class and `sss` indicates subclass. Any SQLSTATE that has a class of:

- '01', is a warning.
- 'HY', is generated by the CLI or ODBC driver.
- 'IM', is generated by the ODBC driver manager.

Note: Versions of CLI before Version 5 returned SQLSTATEs with a class of 'S1' rather than 'HY'. To force the CLI driver to return 'S1' SQLSTATEs, the application must set the environment attribute SQL_ATTR_ODBC_VERSION to the value SQL_OV_ODBC2.

CLI SQLSTATEs include both additional IBM defined SQLSTATEs that are returned by the database server, and CLI defined SQLSTATEs for conditions that are not defined in the ODBC version 3 and ISO SQL/CLI specifications. This allows for the maximum amount of diagnostic information to be returned. When running applications in an ODBC environment, it is also possible to receive ODBC defined SQLSTATEs.

Follow these guidelines for using SQLSTATEs within your application:

- Always check the function return code before calling `SQLGetDiagRec()` to determine if diagnostic information is available.
- Use the SQLSTATEs rather than the native error code.
- To increase your application's portability, only build dependencies on the subset of CLI SQLSTATEs that are defined by the ODBC version 3 and ISO SQL/CLI specifications, and return the additional ones as information only. A dependency in an application is a logic flow decision based on specific SQLSTATEs.

  Note: It might be useful to build dependencies on the class (the first 2 characters) of the SQLSTATEs.
- For maximum diagnostic information, return the text message along with the SQLSTATE (if applicable, the text message will also include the IBM defined SQLSTATE). It is also useful for the application to print out the name of the function that returned the error.

- Ensure that the string allocated for the SQLSTATE includes space for the null termination character returned by CLI.

The code segment from `utilcli.c` shows how diagnostic information, such as SQLSTATEs, can be retrieved and displayed:

```
void HandleDiagnosticsPrint(SQLSMALLINT htype, /* handle type identifier */
                            SQLHANDLE hndl /* handle */ )
{
  SQLCHAR message[SQL_MAX_MESSAGE_LENGTH + 1];
  SQLCHAR sqlstate[SQL_SQLSTATE_SIZE + 1];
  SQLINTEGER sqlcode;
  SQLSMALLINT length, i;

  i = 1;

  /* get multiple field settings of diagnostic record */
  while (SQLGetDiagRec(htype,
                       hndl,
                       i,
                       sqlstate,
                       &sqlcode,
                       message,
                       SQL_MAX_MESSAGE_LENGTH + 1,
                       &length) == SQL_SUCCESS)
  {
    printf("\n  SQLSTATE          =
    printf("  Native Error Code =
    printf("
    i++;
  }

  printf("------------------------\n");
}
```

You can use the CLI/ODBC trace facility to gain a better understanding of how your application calls DB2, including any errors that might occur.

Return codes for compound SQL (CLI) in CLI applications

Return codes are generated on the call to `SQLExecute()` or `SQLExecDirect()` for the END COMPOUND statement. The following lists the return codes for ATOMIC and NOT ATOMIC compound statements:

ATOMIC

- SQL_SUCCESS: all substatements have executed without any warnings or errors.
- SQL_SUCCESS_WITH_INFO: all substatements executed successfully with one or more warnings. Call `SQLGetDiagRec()` or `SQLGetDiagField()` to retrieve additional information about the error or warning. The handle used by `SQLGetDiagRec()` or `SQLGetDiagField()` must be the same one used to process the BEGIN COMPOUND and END COMPOUND statements.
- SQL_NO_DATA_FOUND: BEGIN COMPOUND and END COMPOUND statements executed without any substatements, or none of the substatements affected any rows.
- SQL_ERROR: one or more substatements failed and all substatements were rolled back.

NOT ATOMIC

- SQL_SUCCESS: all substatements executed without any errors.

- SQL_SUCCESS_WITH_INFO: the COMPOUND statement executed with one or more warnings returned by one or more substatements. Call `SQLGetDiagRec()` or `SQLGetDiagField()` to retrieve additional information about the error or warning. The handle used by `SQLGetDiagRec()` or `SQLGetDiagField()` must be the same one used to process the BEGIN COMPOUND and END COMPOUND statements.
- SQL_NO_DATA_FOUND: the BEGIN COMPOUND and END COMPOUND statements executed without any substatements, or none of the substatements affected any rows.
- SQL_ERROR: the COMPOUND statement failed. At least one substatement returned an error. Examine the SQLCA to determine which statements failed.

Chapter 3. CLI/ODBC configuration keywords listing by category

Categorized list of CLI configuration keywords

The CLI/ODBC configuration keywords are divided into the listed categories:

- Compatibility configuration keywords
- Data source configuration keywords
- Data type configuration keywords
- Enterprise configuration keywords
- Environment configuration keywords
- File DSN configuration keywords
- Optimization configuration keywords
- Service configuration keywords
- Static SQL configuration keywords
- Transaction configuration keywords

While most CLI/ODBC configuration keywords can be set in the `db2cli.ini` initialization file or by providing the keyword information in the connection string to `SQLDriverConnect()` function, the **`Trusted_Connection`** Keyword can be set only with `SQLDriverConnect()`.

Compatibility configuration keywords

Use the compatibility configuration keywords to define DB2 behavior. You can set the compatibility configuration keywords to ensure that other applications are compatible with DB2.

- "Trusted_Connection CLI/ODBC configuration keyword" on page 420(use `SQLDriverConnect()` to set this keyword)
- "TimestampTruncErrToWarning CLI/ODBC configuration keyword" on page 407

Data source configuration keywords

General keywords associated with a data source configuration.

- "DBAlias CLI/ODBC configuration keyword" on page 352
- "ClientEncAlg CLI/ODBC configuration keyword" on page 339
- "PWD CLI/ODBC configuration keyword" on page 380
- "UID CLI/ODBC configuration keyword" on page 422

Data type configuration keywords

Use the data type configuration keywords to define how DB2 reports and handles various data types.

- "BitData CLI/ODBC configuration keyword" on page 335
- "CurrentImplicitXMLParseOption CLI/ODBC configuration keyword" on page 345
- "DateTimeStringFormat CLI/ODBC configuration keyword" on page 353
- "DecimalFloatRoundingMode CLI/ODBC configuration keyword" on page 354
- "FloatPrecRadix CLI/ODBC configuration keyword" on page 361
- "Graphic CLI/ODBC configuration keyword" on page 363
- "LOBMaxColumnSize CLI/ODBC configuration keyword" on page 368
- "LongDataCompat CLI/ODBC configuration keyword" on page 369
- "MapBigintCDefault CLI/ODBC configuration keyword" on page 370
- "MapCharToWChar CLI/ODBC configuration keyword" on page 370
- "MapDateCDefault CLI/ODBC configuration keyword" on page 371
- "MapDateDescribe CLI/ODBC configuration keyword" on page 371
- "MapDecimalFloatDescribe CLI/ODBC configuration keyword" on page 372
- "MapGraphicDescribe CLI/ODBC configuration keyword" on page 373
- "MapTimeCDefault CLI/ODBC configuration keyword" on page 373
- "MapTimeDescribe CLI/ODBC configuration keyword" on page 374
- "MapTimestampCDefault CLI/ODBC configuration keyword" on page 374
- "MapTimestampDescribe CLI/ODBC configuration keyword" on page 375
- "MapXMLCDefault CLI/ODBC configuration keyword" on page 376
- "MapXMLDescribe CLI/ODBC configuration keyword" on page 376
- "OleDbReturnCharAsWChar CLI/ODBC configuration keyword" on page 378
- "PromoteLONGVARtoLOB CLI/ODBC configuration keyword" on page 389
- "XMLDeclaration CLI/ODBC configuration keyword" on page 425

Enterprise configuration keywords

Use the enterprise configuration keywords to maximize the efficiency of connections to large databases.

- "ConnectNode CLI/ODBC configuration keyword" on page 342
- "CurrentPackagePath CLI/ODBC configuration keyword" on page 346
- "CurrentPackageSet CLI/ODBC configuration keyword" on page 347

Environment configuration keywords

Use the environment configuration keywords to define environment-specific settings, such as the location of various files on the server and client machines.

File DSN configuration keywords

Use the File DSN configuration keywords to set the TCP/IP settings for a file DSN connection.

- "SSLClientKeystash CLI/ODBC configuration keyword" on page 400
- "TargetPrincipal CLI/ODBC configuration keyword" on page 406

Optimization configuration keywords

Use the optimization configuration keywords to speed up and reduce the amount of network flow between the IBM Data Server Driver for ODBC and CLI and the server.

- "AllowGetDataLOBReaccess CLI/ODBC configuration keyword" on page 327
- "AppendForFetchOnly CLI/ODBC configuration keyword" on page 329
- "AppUsesLOBLocator CLI/ODBC configuration keyword" on page 329
- "BlockForNRows CLI/ODBC configuration keyword" on page 335
- "BlockLobs CLI/ODBC configuration keyword" on page 336
- "ClientBuffersUnboundLOBS CLI/ODBC configuration keyword" on page 338
- "ColumnwiseMRI CLI/ODBC configuration keyword" on page 341
- "ConcurrentAccessResolution CLI/ODBC configuration keyword" on page 341
- "CurrentMaintainedTableTypesForOpt CLI/ODBC configuration keyword" on page 345
- "DB2Degree CLI/ODBC configuration keyword" on page 349
- "DB2Explain CLI/ODBC configuration keyword" on page 350
- "DB2NETNamedParam CLI/ODBC configuration keyword" on page 351
- "DB2Optimization CLI/ODBC configuration keyword" on page 351
- "DescribeInputOnPrepare CLI/ODBC configuration keyword" on page 356
- "DescribeOutputLevel CLI/ODBC configuration keyword" on page 357
- "FET_BUF_SIZE CLI/ODBC configuration keyword" on page 361
- "GetDataLobNoTotal CLI/ODBC configuration keyword" on page 362
- "KeepDynamic CLI/ODBC configuration keyword" on page 366
- "LOBCacheSize CLI/ODBC configuration keyword" on page 367
- "LOBFileThreshold CLI/ODBC configuration keyword" on page 368
- "LockTimeout CLI/ODBC configuration keyword" on page 369
- "MaxLOBBlockSize CLI/ODBC configuration keyword" on page 377
- "OptimizeForNRows CLI/ODBC configuration keyword" on page 380
- "Reopt CLI/ODBC configuration keyword" on page 391
- "ReturnAliases CLI/ODBC configuration keyword" on page 395
- "SkipTrace CLI/ODBC configuration keyword" on page 399
- "StmtConcentrator CLI/ODBC configuration keyword" on page 403
- "StreamGetData CLI/ODBC configuration keyword" on page 404
- "StreamPutData CLI/ODBC configuration keyword" on page 404
- "Underscore CLI/ODBC configuration keyword" on page 423

Service configuration keywords

Use the service configuration keywords to help in troubleshooting problems with CLI/ODBC connections. Programmers can also use service configuration keywords to gain a better understanding of how their CLI programs are translated into calls to the server.

- "AppendAPIName CLI/ODBC configuration keyword" on page 329

- "AppendRowColToErrorMessage CLI/ODBC configuration keyword" on page 330
- "IgnoreWarnings CLI/ODBC configuration keyword" on page 365
- "IgnoreWarnList CLI/ODBC configuration keyword" on page 364
- "LoadXAInterceptor CLI/ODBC configuration keyword" on page 368
- "Patch1 CLI/ODBC configuration keyword" on page 381
- "Patch2 CLI/ODBC configuration keyword" on page 384
- "ReportRetryErrorsAsWarnings CLI/ODBC configuration keyword" on page 392
- "RetryOnError CLI/ODBC configuration keyword" on page 394
- "ProgramID CLI/ODBC configuration keyword" on page 388
- "ProgramName CLI/ODBC configuration keyword" on page 388
- "Trace CLI/ODBC configuration keyword" on page 408
- "TraceAPIList CLI/ODBC configuration keyword" on page 409
- "TraceAPIList! CLI/ODBC configuration keyword" on page 411
- "TraceComm CLI/ODBC configuration keyword" on page 413
- "TraceErrImmediate CLI/ODBC configuration keyword" on page 413
- "TraceFileName CLI/ODBC configuration keyword" on page 414
- "TraceFlush CLI/ODBC configuration keyword" on page 415
- "TraceFlushOnError CLI/ODBC configuration keyword" on page 415
- "TraceLocks CLI/ODBC configuration keyword" on page 416
- "TracePathName CLI/ODBC configuration keyword" on page 417
- "TracePIDList CLI/ODBC configuration keyword" on page 416
- "TracePIDTID CLI/ODBC configuration keyword" on page 417
- "TraceRefreshInterval CLI/ODBC configuration keyword" on page 418
- "TraceStmtOnly CLI/ODBC configuration keyword" on page 419
- "TraceTime CLI/ODBC configuration keyword" on page 419
- "TraceTimestamp CLI/ODBC configuration keyword" on page 420
- "WarningList CLI/ODBC configuration keyword" on page 425

Static SQL configuration keywords

Use the static SQL configuration keywords when running static SQL statements in CLI/ODBC applications.

- "StaticCapFile CLI/ODBC configuration keyword" on page 402
- "StaticLogFile CLI/ODBC configuration keyword" on page 402
- "StaticMode CLI/ODBC configuration keyword" on page 402
- "StaticPackage CLI/ODBC configuration keyword" on page 403

Transaction configuration keywords

Use the transaction configuration keywords to control and speed up SQL statements that are used in the application.

- "ArrayInputChain CLI/ODBC configuration keyword" on page 331
- "AsyncEnable CLI/ODBC configuration keyword" on page 331
- "AutoCommit CLI/ODBC configuration keyword" on page 334
- "ClientAcctStr CLI/ODBC configuration keyword" on page 337
- "ClientApplName CLI/ODBC configuration keyword" on page 338

db2cli.ini initialization file

The CLI/ODBC initialization file (`db2cli.ini`) contains various keywords and values that can be used to configure the behavior of CLI and the applications using it.

The keywords are associated with the database alias name, and affect all CLI and ODBC applications that access the database.

The `db2cli.ini.sample` sample configuration file is included to help you get started. You can create a `db2cli.ini` file that is based on the `db2cli.ini.sample` file and that is stored in the same location. The location of the sample configuration file depends on your driver type and platform.

For IBM Data Server Client, IBM Data Server Runtime Client, or IBM Data Server Driver Package, the sample configuration file is created in one of the following paths:

- On AIX®, HP-UX, Linux, or Solaris operating systems: *installation_path*`/cfg`
- On Windows XP and Windows Server 2003: `C:\Documents and Settings\All Users\Application Data\IBM\DB2\`*driver_copy_name*`\cfg`
- On Windows Vista and Windows Server 2008: `C:\ProgramData\IBM\DB2\`*driver_copy_name*`\cfg`

For example, if you use IBM Data Server Driver Package for Windows XP, and the data server driver copy name is `IBMDBCL1`, then the `db2cli.ini.sample` file is created in the `C:\Documents and Settings\All Users\Application Data\IBM\DB2\IBMDBCL1\cfg` directory.

For IBM Data Server Driver for ODBC and CLI, the sample configuration file is created in one of the following paths:

- On AIX, HP-UX, Linux, or Solaris operating systems: *installation_path*`/cfg`
- On Windows : *installation_path*`\cfg`

  where *installation_path* is the file path where driver files are extracted.

For example, if you use IBM Data Server Driver for ODBC and CLI for Windows Vista, and the driver is installed in the `C:\IBMDB2\CLIDRIVER\V97FP3` directory, then the `db2cli.ini.sample` file is created in the `C:\IBMDB2\CLIDRIVER\V97FP3\cfg` directory.

When the ODBC Driver Manager is used to configure a user DSN on Windows operating systems, the `db2cli.ini` file is created in `Documents and Settings\User Name` where *User Name* represents the name of the user directory.

You can use the environment variable **DB2CLIINIPATH** to specify a different location for the db2cli.ini file.

The configuration keywords enable you to:

- Configure general features such as data source name, user name, and password.
- Set options that will affect performance.
- Indicate query parameters such as wild card characters.
- Set patches or work-arounds for various ODBC applications.
- Set other, more specific features associated with the connection, such as code pages and IBM GRAPHIC data types.
- Override default connection options specified by an application. For example, if an application requests Unicode support from the CLI driver by setting the SQL_ATTR_ANSI_APP connection attribute, then setting **DisableUnicode**=1 in the db2cli.ini file will force the CLI driver not to provide the application with Unicode support.

  Note: If the CLI/ODBC configuration keywords set in the db2cli.ini file conflict with keywords in the SQLDriverConnect() connection string, then the SQLDriverConnect() keywords will take precedence.

The db2cli.ini initialization file is an ASCII file which stores values for the CLI configuration options. A sample file is included to help you get started. While most CLI/ODBC configuration keywords are set in the db2cli.ini initialization file, some keywords are set by providing the keyword information in the connection string to SQLDriverConnect() instead.

There is one section within the file for each database (data source) the user wishes to configure. If needed, there is also a common section that affects all database connections.

Only the keywords that apply to all database connections through the CLI/ODBC driver are included in the COMMON section. This includes the following keywords:

- **CheckForFork**
- **DiagPath**
- **DisableMultiThread**
- **JDBCTrace**
- **JDBCTraceFlush**
- **JDBCTracePathName**
- **QueryTimeoutInterval**
- **ReadCommonSectionOnNullConnect**
- **Trace**
- **TraceComm**
- **TraceErrImmediate**
- **TraceFileName**
- **TraceFlush**
- **TraceFlushOnError**
- **TraceLocks**
- **TracePathName**
- **TracePIDList**

- **TracePIDTID**
- **TraceRefreshInterval**
- **TraceStmtOnly**
- **TraceTime**
- **TraceTimeStamp**

All other keywords are to be placed in the database specific section.

Note: Configuration keywords are valid in the COMMON section, however, they will apply to all database connections.

The COMMON section of the `db2cli.ini` file begins with:

```
[COMMON]
```

Before setting a common keyword it is important to evaluate its affect on all CLI/ODBC connections from that client. A keyword such as **TRACE**, for example, will generate information about all CLI/ODBC applications connecting to DB2 on that client, even if you are intending to troubleshoot only one of those applications.

Each database specific section always begins with the name of the data source name (DSN) between square brackets:

[*data source name***]**

This is called the *section header*.

The parameters are set by specifying a keyword with its associated keyword value in the form:

KeywordName =*keywordValue*

- All the keywords and their associated values for each database must be located under the database section header.
- If the database-specific section does not contain a **DBAlias** keyword, the data source name is used as the database alias when the connection is established. The keyword settings in each section apply only to the applicable database alias.
- The keywords are not case sensitive; however, their values can be if the values are character based.
- If a database is not found in the .INI file, the default values for these keywords are in effect.
- Comment lines are introduced by having a semicolon in the first position of a new line.
- Blank lines are permitted.
- If duplicate entries for a keyword exist, the first entry is used (and no warning is given).

The following sample .INI file contains two database alias sections:

```
; This is a comment line.
[MYDB22]
AutoCommit=0
TableType="'TABLE','SYSTEM TABLE'"

; This is another comment line.
[MYDB2MVS]
CurrentSQLID=SAAID
TableType="'TABLE'"
SchemaList="'USER1',CURRENT SQLID,'USER2'"
```

Although you can edit the db2cli.ini file manually on all platforms, it is recommended that you use the **UPDATE CLI CONFIGURATION** command if it is available. You must add a blank line after the last entry if you manually edit the db2cli.ini file.

AllowGetDataLOBReaccess CLI/ODBC configuration keyword

Specifies whether the application can call SQLGetData() for previously accessed LOB columns when querying database servers that support Dynamic Data Format.

db2cli.ini keyword syntax:

AllowGetDataLOBReaccess = **0** | **1**

Default setting:

Do not allow calls to SQLGetData() for previously accessed LOB columns when querying database servers that support Dynamic Data Format.

Usage notes:

This keyword only affects connections to database servers that support Dynamic Data Format, also known as progressive streaming. The default setting of 0 does not allow applications to call SQLGetData() for previously accessed LOB columns. Specify 1 to allow applications to call SQLGetData() for previously accessed LOB columns.

Note that when the keyword is set to 1 to allow re-access to LOB columns, some resources on the server might not be freed upon completion of SQLGetData().

If the server does not support Dynamic Data Format, this keyword has no effect and calls to SQLGetData() for previously accessed LOB columns are allowed.

A similar keyword exists called AllowInterleavedGetData that allows applications to call SQLGetData() for previously accessed LOB columns and maintain the data offset position from the previous call to SQLGetData() when querying data servers that support Dynamic Data Format. If both AllowGetDataLOBReaccess and AllowInterleavedGetData are set for a given connection or statement, the AllowInterleavedGetData setting takes precedence over AllowGetDataLOBReaccess.

AllowInterleavedGetData CLI/ODBC configuration keyword

Specifies whether the application can call SQLGetData() for previously accessed LOB columns and maintain the data offset position from the previous call to SQLGetData() when querying data servers that support Dynamic Data Format.

db2cli.ini keyword syntax:

AllowInterleavedGetData = **0** | **1**

Default setting:

Do not allow calls to SQLGetData() for previously accessed LOB columns when querying database servers that support Dynamic Data Format.

Equivalent environment or connection attribute:

SQL_ATTR_ALLOW_INTERLEAVED_GETDATA

Usage notes:

This keyword affects only connections to database servers that support Dynamic Data Format, also known as progressive streaming. The default setting of 0 does not allow applications to call SQLGetData() for previously accessed LOB columns. Specify 1 to allow applications to call SQLGetData()

for previously accessed LOB columns and start reading LOB data from where the application stopped reading during the previous read.

Note that when the keyword is set to 1 to allow re-access to LOB columns, some resources on the server might not be freed upon completion of SQLGetData().

If the server does not support Dynamic Data Format, this keyword has no effect, and calls to SQLGetData() for previously accessed LOB columns are allowed.

A similar keyword exists called AllowGetDataLOBReaccess that allows applications to call SQLGetData() for previously accessed LOB columns. However, if the AllowGetDataLOBReaccess keyword is used, data position and offset information is not maintained. When the LOB column is re-accessed after interleaving, SQLGetData() starts reading data from the beginning for that LOB data column. If both AllowGetDataLOBReaccess and AllowInterleavedGetData are set for a given connection or statement, the AllowInterleavedGetData setting takes precedence over AllowGetDataLOBReaccess.

AltHostName CLI/ODBC configuration keyword

Specifies the alternate host name to be used if the primary server specified by HOSTNAME cannot be contacted (Client Reroute.)

db2cli.ini keyword syntax:

AltHostName = *fully qualified alternate host name | IP address of node*

Usage notes:

This can be set in the [Data Source] section of the db2cli.ini file for the given data source, or in a connection string.

This parameter specifies a fully qualified host name or the IP address of the node where the alternate server for the database resides.

If the primary server returns alternate server information, it will override this AltHostName setting. However, this keyword is read only. That means the db2cli.ini will not be updated with the alternate server information received from the primary server.

AltPort CLI/ODBC configuration keyword

Specifies the alternate port to be used if the primary server specified by HOSTNAME and PORT cannot be contacted (Client Reroute.)

db2cli.ini keyword syntax:

AltPort = *port number*

Usage notes:

This can be set in the [Data Source] section of the db2cli.ini file for the given data source, or in a connection string.

This parameter specifies the port number of the alternate server of the database manager instance where the alternate server for the database resides.

If the primary server returns alternate server information, it will override this AltPort setting. However, this keyword is read only. That means the db2cli.ini will not be updated with the alternate server information received from the primary server.

AppUsesLOBLocator CLI/ODBC configuration keyword

Specifies whether applications use LOB locators.

db2cli.ini keyword syntax:
AppUsesLOBLocator = **0** | **1**

Default setting:
Applications are using LOB locators.

Equivalent connection or statement attribute:
SQL_ATTR_APP_USES_LOB_LOCATOR

Usage notes:
The default setting of 1 indicates that applications are using LOB locators. For applications that do not use LOB locators and are querying data on a server that supports Dynamic Data Format, also known as progressive streaming, specify 0 to indicate that LOB locators are not used and allow the return of LOB data to be optimized.

This keyword is ignored for stored procedure result sets.

If the keyword is set to 0 and an application binds a LOB locator to a result set using `SQLBindCol()`, an Invalid conversion error will be returned by the `SQLFetch()` function.

AppendAPIName CLI/ODBC configuration keyword

Appends the CLI/ODBC function name which generated an error to the error message text.

db2cli.ini keyword syntax:
AppendAPIName = **0** | **1**

Default setting:
Do NOT display CLI function name.

Usage notes:

The CLI function (API) name that generated an error is appended to the error message retrieved using `SQLGetDiagRec()` or `SQLError()`. The function name is enclosed in curly braces { }.

For example,

```
[IBM][CLI Driver]" CLIxxxx: < text >
SQLSTATE=XXXXX {SQLGetData}"
```

- 0 = do NOT append CLI function name (default)
- 1 = append the CLI function name

This keyword is only useful for debugging.

AppendForFetchOnly CLI/ODBC configuration keyword

Specifies whether the clause FOR FETCH ONLY is appended to READ-ONLY SQL statements.

db2cli.ini keyword syntax:
AppendForFetchOnly = **0** | **1**

Default setting:
The keyword is not set by default. CLI appends the "FOR FETCH ONLY" clause only when connected to certain server types.

Equivalent connection attribute:
SQL_ATTR_APPEND_FOR_FETCH_ONLY

Usage notes:
By default, CLI appends the "FOR FETCH ONLY" clause to read SELECT statements when connected to DB2 for z/OS or DB2 for i databases.

This keyword allows an application to control when CLI appends the "FOR FETCH ONLY" clause, for example, in a situation where an application is binding the CLI packages using different bind BLOCKING options (for example, BLOCKING UNAMBIG) and wants to suppress the blocking in order to keep positioned on a given row.

To change the default CLI behavior, the keyword can be set as follows:

- 0: CLI never appends the "FOR FETCH ONLY" clause to read SELECT statements regardless of the server type it is connecting to.
- 1: CLI always appends the "FOR FETCH ONLY" clause to read SELECT statements regardless of the server type it is connecting to.

AppendRowColToErrorMessage CLI/ODBC configuration keyword

Specifies whether the row and column numbers that generated the error are appended the error message string.

db2cli.ini keyword syntax:
AppendRowColToErrorMessage= **0** | **1**

Default setting:
The default setting of 0 will return the error message string without the row and column numbers.

Usage notes:
Specify 1 to append the row and column number that generated the error to the error message string. The values for row and column numbers are only appended when DB2 CLI is able to apply a row or column number to the problem.

The row or column numbers appended to error messages are the same positive values that would be returned if an application called `SQLGetDiagField()` with the DiagIdentifier argument as SQL_DIAG_ROW_NUMBER or SQL_DIAG_COLUMN_NUMBER. When AppendRowColToErrorMessage is set to 1, errors returned from calls to `SQLGetDescField()`, `SQLGetDescRec()` or `SQLError()` will have these row or column numbers appended with the following format: Row=<r>, Col=<c>, if they can be determined.

For example, the default text for error CLI0111E is as follows:

```
[IBM][CLI Driver] CLI0111E  Numeric value out of range. SQLSTATE=22003
```

Specifying 1 to append the row and column number will return the following text for error CLI0111E:

```
[IBM][CLI Driver] CLI0111E  Numeric value out of range.
   SQLSTATE=22003 {Row=2,Col=1}
```

Note: It is also possible for an error to be returned with only a row number.

ArrayInputChain CLI/ODBC configuration keyword

Enables array input without needing pre-specified size and memory allocation requirements of normal array input.

db2cli.ini keyword syntax:

ArrayInputChain = **-1** | **0** | **<positive integer>**

Default setting:

Normal input array is enabled, where the array and its size must be specified before the corresponding `SQLExecute()` call is made.

Usage notes:

By default, array input (where an array of values is bound to an input parameter) requires the array and its size to be specified before the corresponding `SQLExecute()` function is called. An application, however, may not know the array size in advance, or the array size may be too large for the application to allocate from its pool of available memory. Under these circumstances, the application can set ArrayInputChain=-1 and use the SQL_ATTR_CHAINING_BEGIN and SQL_ATTR_CHAINING_END statement attributes to enable chaining, which allows array input without the pre-specified size and memory requirements of normal array input.

To enable chaining:

1. Set the keyword ArrayInputChain = -1.
2. Prepare and bind input parameters to the SQL statement.
3. Set the SQL_ATTR_CHAINING_BEGIN statement attribute with `SQLSetStmtAttr()`.
4. Update the bound parameters with input data and call `SQLExecute()`.
5. Repeat Step 4 for as many rows as there are in the input array.
6. Set the SQL_ATTR_CHAINING_END statement attribute with `SQLSetStmtAttr()` after the last row in the array has been processed according to Step 4.

The effect of completing these steps will be the same as if normal array input had been used.

Setting ArrayInputChain=0 (the default value) turns this array input feature off. ArrayInputChain can also be set to any positive integer which sets the array size to use for the input array.

Restriction: DB2 CLI does not support array input chaining for compound SQL (compiled) or compound SQL (inlined) statements.

AsyncEnable CLI/ODBC configuration keyword

Enables or disables the ability to execute queries asynchronously.

db2cli.ini keyword syntax:

AsyncEnable = **0** | **1**

Default setting:

Queries can be executed asynchronously.

Usage notes:

This option allows you to enable or disable support that allows queries to

execute asynchronously. This only benefits applications that were written to take advantage of this feature by setting the SQL_ATTR_ASYNC_ENABLE attribute using `SQLSetStmtAttr()` or `SQLSetConnectAttr()`.

- 0 = Queries are not executed asynchronously
- 1 = Allow queries to be executed asynchronously. The application must also enable the asynchronous functionality by setting SQL_ATTR_ASYNC_ENABLE using `SQLSetStmtAttr()` or `SQLSetConnectAttr()`. (default)

Once a function has been called asynchronously, only the original function, `SQLAllocHandle()`, `SQLCancel()`, `SQLSetStmtAttr()`, `SQLGetDiagField()`, `SQLGetDiagRec()`, or `SQLGetFunctions()` can be called on the statement handle, until the original function returns a code other than SQL_STILL_EXECUTING. Any other function called on any other statement handle under the same connection returns SQL_ERROR with an SQLSTATE of HY010 (Function sequence error).

Attach CLI/ODBC configuration keyword

Specifies whether to attach to the server instance. You can specify this keyword in the connection string or set it in the `db2cli.ini` or `db2dsdriver.cfg` file.

db2cli.ini keyword syntax:

ATTACH = **TRUE** | **FALSE**

Default setting:

The SQLDriverConnect() function connects to the specified database.

Equivalent environment or connection attribute:

N/A

Usage notes:

When you set the keyword to TRUE, the SQLDriverConnect() function does not connect to a database but instead connects to the specified server instance.

To establish a connection with a DB2 server instance for Linux, UNIX, and Windows remote server, the CLI application must specify values for the Hostname, Port, UID, PWD, and Protocol along with setting the keyword to `TRUE`.

Any value other than `TRUE` that you assign to the **`ATTACH`** keyword is treated as `FALSE`.

Examples:

The following example shows a specification of the keyword in the `db2cli.ini` file:

```
ATTACH=TRUE
```

The following example shows specifications of the keyword in the `db2dsdriver.cfg` file:

```
<configuration>
   <dsncollection>
        <dsn
alias="db2dsn01",name="db2db01",host="server1.mynet.com",
port="50001">
           <parameter name="ATTACH" value="TRUE"/>
        </dsn>
    </dsncollection>
```

```
    <databases>
        <database name="sample", host="serv1.mynet.com",
port="50001">
          <parameter name="ATTACH" value="TRUE"/>
      </database>
   </databases>
</configuration>
```

Version information

Last update
This topic was last updated for IBM DB2 Version 9.7, Fix Pack 3.

IBM Data Server Client
Supported in IBM DB2 Database for Linux, UNIX, and Windows

Authentication CLI/ODBC configuration keyword

Specifies the type of authentication to be used with file DSN or DSN-less connectivity.

db2cli.ini keyword syntax:
Authentication = **CERTIFICATE** | **SERVER** | **SERVER_ENCRYPT** | **SERVER_ENCRYPT_AES** | **DATA_ENCRYPT** | **KERBEROS** | **GSSPLUGIN**

Default setting:
SERVER

Usage notes:
This can be set in the [Data Source] section of the db2cli.ini file for the given data source, or in a connection string.

When you set this option, you must also set the following options:

- **Database**
- **Protocol**.

If **Protocol**=IPC, you need to set the following option as well:

- **Instance**.

If **Protocol**=TCPIP, you need to set the following options as well:

- **Port**
- **Hostname**.

If Kerberos is specified, then the **KRBPlugin** may also be optionally specified. If **KRBPlugin** is not specified, the default plug-in IBMkrb5 will be used.

Starting in DB2 Version 9.7 Fix Pack 6 and later, CERTIFICATE authentication is available for connection to DB2 for z/OS Version 10 with APAR PM53450 and later. The CERTIFICATE authentication type is supported starting in DB2 Version 9.7 Fix Pack 6. This authentication type allows you to use SSL client authentication without the need of providing database passwords on the database client. When certificate-based authentication is configured to supply authentication information, a password cannot be specified in any other way (as in the db2dsdriver.cfg configuration file, in the db2cli.ini configuration file, or in the connection string). If CERTIFICATE is specified, then the new label parameter SSLCLientLabel must also be specified in the CLI configuration file, db2cli.ini, or in the data server driver configuration file, db2dsdriver.cfg.

AutoCommit CLI/ODBC configuration keyword

Specifies whether the application commits each statement by default.

db2cli.ini keyword syntax:
AutoCommit = **1** | **0**

Default setting:
Each statement is treated as a single, complete transaction.

Equivalent connection attribute:
SQL_ATTR_AUTOCOMMIT

Usage notes:

To be consistent with ODBC, CLI defaults with AutoCommit on, which means each statement is treated as a single, complete transaction. This keyword can provide an alternative default, but will only be used if the application does not specify a value for SQL_ATTR_AUTOCOMMIT.

- 1 = SQL_ATTR_AUTOCOMMIT_ON (default)
- 0 = SQL_ATTR_AUTOCOMMIT_OFF

Note: Most ODBC applications assume the default of AutoCommit to be on. Extreme care must be used when overriding this default during runtime as the application may depend on this default to operate properly.

This keyword also allows you to specify whether autocommit should be enabled in a Distributed Unit of Work (DUOW) environment. If a connection is part of a coordinated Distributed Unit of Work, and AutoCommit is not set, the default does not apply; implicit commits arising from autocommit processing are suppressed. If AutoCommit is set to 1, and the connection is part of a coordinated Distributed Unit of Work, the implicit commits are processed. This may result in severe performance degradation, and possibly other unexpected results elsewhere in the DUOW system. However, some applications may not work at all unless this is enabled.

A thorough understanding of the transaction processing of an application is necessary, especially applications written by a third party, before applying it to a DUOW environment.

BIDI CLI/ODBC configuration keyword

Specifies the BIDI code page when connected to a DB2 for z/OS.

db2cli.ini keyword syntax:
BIDI = *code page*

Usage notes:
This can be set in the `[Data Source]` section of the `db2cli.ini` file for the given data source, or in a connection string.

When you set this option, you must also set the listed options:

- Database
- Protocol=TCPIP
- Hostname
- Port

BitData CLI/ODBC configuration keyword

Specifies whether binary data types are reported as binary or character data types.

db2cli.ini keyword syntax:

BitData = **1** | **0**

Default setting:

Report FOR BIT DATA and BLOB data types as binary data types.

Usage notes:

This option allows you to specify whether ODBC binary data types (SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY, and SQL_BLOB), are reported as binary type data. IBM DBMSs support columns with binary data types by defining CHAR, VARCHAR, and LONG VARCHAR columns with the FOR BIT DATA attribute. DB2 Database for Linux, UNIX, and Windows will also support binary data via the BLOB data type (in this case it is mapped to a CLOB data type).

Only set BitData = 0 if you are sure that all columns defined as FOR BIT DATA or BLOB contain only character data, and the application is incapable of displaying binary data columns.

- 1 = report FOR BIT DATA and BLOB data types as binary data types (default).
- 0 = report FOR BIT DATA and BLOB data types as character data types.

BlockForNRows CLI/ODBC configuration keyword

Specifies the number of rows of data to be returned in a single fetch.

db2cli.ini keyword syntax:

BlockForNRows = <positive integer>

Default setting:

The server returns as many rows as can fit in a query block in a single fetch request.

Usage notes:

The BlockForNRows keyword controls the number of rows of data that are returned to the client in a single fetch request. If BlockForNRows is not specified (the default setting), then as many rows of non-LOB data as can fit in a query block are returned from the server. If the result set contains LOB data, then the behavior BlockForNRows yields can be affected by the BlockLobs CLI/ODBC configuration keyword and the server's support for blocking of result sets returning LOB data types.

All LOB data associated with rows that fit completely within a single query block are returned in a single fetch request if:

- BlockForNRows is not specified,
- BlockLobs is set to 1 and
- the server supports blocking of result sets returning LOB data types.

LOB data is described here as being associated with a row, because the LOB data of a result set is itself not contained in the row. Instead, the row contains a reference to the actual LOB data.

If BlockForNRows is set to a positive integer n, then n rows of data will be returned in a single fetch request. If the result set contains LOB data and the server supports blocking of result sets returning LOB data types, then the LOB data that corresponds to the n rows of data will also be returned in the single fetch request. If the result set contains LOB data, but the server does not support blocking of result sets returning LOB data types, then only one row of data, including the LOB data, will be returned in a single fetch request.

BlockLobs CLI/ODBC configuration keyword

Enables LOB blocking fetch against servers that support LOB blocking.

db2cli.ini keyword syntax:

BlockLobs = **0** | **1**

Default setting:

Blocking of result sets returning LOB data types is disabled.

Equivalent statement attribute:

SQL_ATTR_BLOCK_LOBS

Usage notes:

Setting **BlockLobs** to 1 enables all of the LOB data associated with rows that fit completely within a single query block to be returned in a single fetch request, if the server supports LOB blocking. CLI clients which enable **BlockLobs** = 1 and bind the LOB values directly to buffers can show an increase in memory consumption depending on the amount of data retrieved for one request compared to previous releases. LOB data is described here as being associated with a row, because the LOB data of a result set is itself not contained in the row. Instead, the row contains a reference to the actual LOB data. Therefore, with blocking of result sets returning LOB data types, any rows of the result set that fit completely within the query block (where each row consists of non-LOB data, since LOB data is not stored directly in the row), will have their associated LOB data returned from the server, if the server supports blocking of result sets returning LOB data types.

If the server does not support cursor blocking with LOB columns, then only one row of LOB data is returned in a single fetch request and the **BlockLobs** value is ignored. While DB2 Database for Linux, UNIX, and Windows does support cursor blocking with LOB columns, other servers might not.

DB2 Database for Linux, UNIX, and Windows does not support LOB blocking fetch.

IDS data servers do not support LOB blocking fetch.

CLIPkg CLI/ODBC configuration keyword

Specifies the number of large packages to be generated.

db2cli.ini keyword syntax:

CLIPkg = **3** | **4** | ... | **30**

Default setting:

Three large packages are generated.

Usage notes:

This keyword is used to increase the number of sections for SQL statements in CLI/ODBC applications. If it is used, the administrator should explicitly bind the required bind files with the CLIPkg bind option. For client applications, the `db2cli.ini` file on the client must be updated with this value of CLIPkg. For CLI/JDBC stored procedures, the `db2cli.ini` file on the server (DB2 UDB Version 6.1 or later on UNIX or Intel platforms) must be updated with the same value of CLIPkg.

If the value is NOT an integer between 3 and 30, the default will be used without error or warning.

This setting only applies to large packages (containing 384 sections). The number of small packages (containing 64 sections) is 3 and cannot be changed.

It is recommended that you only increase the number of sections enough to run your application as the packages take up space in the database.

CheckForFork CLI/ODBC configuration keyword

Checks for a forked process for each function call.

db2cli.ini keyword syntax:

0 | **1**

Default setting:

CLI does not check for forked processes.

Usage notes:

CLI assumes that the process will never be forked. The **CheckForFork** keyword must be set to 1 if applications want to fork while connection and statement handles are allocated in order to avoid interfering with the parent process' active connections.

The SQL_ATTR_PROCESSCTL environment attribute can be set to SQL_PROCESSCTL_NOTHREAD option by an application to override the **CheckForFork** keyword for that application.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2 databases.)

ClientAcctStr CLI/ODBC configuration keyword

Sets the client accounting string that is sent to a database.

db2cli.ini keyword syntax:

ClientAcctStr = *accounting string*

Default setting:

None

Applicable when:

Connected to a database using DB2 Connect or DB2 Database for Linux, UNIX, and Windows

Equivalent environment or connection attribute:

SQL_ATTR_INFO_ACCTSTR

Usage notes:

This option allows the CLI application to set the client accounting string that is sent to the database through DB2 Connect or DB2 database products. Applications that do not offer the accounting string by default can take advantage of this keyword to provide this information.

Note the following conditions:

- When the value is being set, some servers might not handle the entire length provided and might truncate the value.
- DB2 for z/OS and OS/390 servers support up to a length of 200 characters.
- To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

ClientApplName CLI/ODBC configuration keyword

Sets the client application name that is sent to a database.

db2cli.ini keyword syntax:
ClientApplName = *application name*

Default setting:
None

Applicable when:
Connected to a database using DB2 Connect or DB2 database products for Linux, UNIX and Windows

Equivalent environment or connection attribute:
SQL_ATTR_INFO_APPLNAME

Usage notes:

This option allows the CLI application to set the client application name that is sent to the database through DB2 Connect or DB2 database products. Applications that do not offer the application name by default can take advantage of this keyword to provide this information.

Note the following conditions:

- When the value is being set, some servers might not handle the entire length provided and might truncate the value.
- DB2 for z/OS and OS/390 servers support up to a length of 32 characters.
- To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

ClientBuffersUnboundLOBS CLI/ODBC configuration keyword

Specifies whether LOB data is fetched instead of the LOB locator for LOB columns that have not been bound to application parameters.

db2cli.ini keyword syntax:
ClientBuffersUnboundLOBS = **0** | **1**

Default setting:
A LOB locator is retrieved instead of the actual LOB data for LOB columns that have not been bound to application parameters.

Usage notes:

By default, when a result set contains a LOB column that has not been bound to an application parameter, CLI will fetch the corresponding LOB locator rather than the LOB data itself. The application must then use the `SQLGetLength()`, `SQLGetPosition()`, and `SQLGetSubString()` CLI functions to retrieve the LOB data. If the application regularly wants to retrieve the LOB data, then this default two-step process is unnecessary and could decrease performance. In this case, set ClientBuffersUnboundLOBS = 1 to force DB2 CLI to fetch the LOB data instead of the LOB locator.

Servers that support Dynamic Data Format, also known as progressive streaming, optimize the return of LOB and XML data depending on the actual length of the data. The LOB and XML data can be returned in its entirety, or as an internal token called a progressive reference. CLI manages progressive reference data retrieval.

For applications that are querying data on a server that supports Dynamic Data Format, setting the LOBCacheSize keyword sets a threshold that is used to determine if the data is returned in its entirety, or as a progressive reference. If the data has a length greater than the LOBCacheSize threshold value, the progressive reference will be returned to CLI to manage, but if the data has a length less than or equal to the LOBCacheSize threshold value, the data will be returned in its entirety. Setting ClientBuffersUnboundLOBS to 1 is equivalent to setting LOBCacheSize to 2147483647 and will force the server to return the data in its entirety rather than as a progressive reference.

ClientEncAlg CLI/ODBC configuration keyword

Specifies the type of encryption algorithm to be used when encrypting user IDs and passwords.

db2cli.ini keyword syntax:

ClientEncAlg = **1** | **2** | **AES**

Default setting:

Any encryption algorithm can be used.

Applicable when:

Connecting to remote databases.

Equivalent environment or connection attribute:

SQL_ATTR_CLIENT_ENCALG

Usage notes:

The values for this keyword are defined as follows:

- `1` - Encrypt the user ID and password using any encryption algorithm.
- `2` - Encrypt the user ID and password using an Advanced Encryption Standard (AES) encryption algorithm.
- `AES` - Equivalent to `2`.

The CLI attribute SQL_ATTR_CLIENT_ENCALG has a similar behavior as the keyword, except when an invalid attribute value is specified an error is returned: CLI0191E Invalid attribute value. The CLI keyword or connection attribute values take precedence over the authentication type specified in the system database directory.

ClientUserID CLI/ODBC configuration keyword

Sets the client user ID that is sent to a database.

db2cli.ini keyword syntax:
ClientUserID = *userid*

Default setting:
None

Applicable when:
Connected to a database using DB2 Connect or DB2 Database for Linux, UNIX, and Windows

Equivalent environment or connection attribute:
SQL_ATTR_INFO_USERID

Usage notes:

This option allows the CLI application to set the client user ID (accounting user ID) that is sent to the database through DB2 Connect or DB2 database products. Applications that do not offer the user ID by default can take advantage of this keyword to provide this information.

Note the following conditions:

- When the value is being set, some servers might not handle the entire length provided and might truncate the value.
- DB2 for z/OS and OS/390 servers support up to a length of 16 characters.
- This user ID is not to be confused with the authentication user ID. This user ID is for identification purposes only and is not used for any authorization.
- To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

ClientWrkStnName CLI/ODBC configuration keyword

Sets the client workstation name that is sent to a database.

db2cli.ini keyword syntax:
ClientWrkStnName = *workstation name*

Default setting:
None

Applicable when:
Connected to a database using DB2 Connect or DB2 Database for Linux, UNIX, and Windows

Equivalent environment or connection attribute:
SQL_ATTR_INFO_WRKSTNNAME

Usage notes:

This option allows the CLI application to set the client workstation name that is sent to the database through DB2 Connect or DB2 database products. Applications that do not offer the client workstation name by default can take advantage of this keyword to provide this information.

Note the following conditions:

- When the value is being set, some servers might not handle the entire length provided and might truncate the value.
- DB2 for z/OS and OS/390 servers support up to a length of 18 characters.

- To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

ColumnwiseMRI CLI/ODBC configuration keyword

Specifies whether array input chaining is converted to column-wise array input for DB2 for z/OS servers.

db2cli.ini keyword syntax:

ColumnwiseMRI = **ON** | **OFF**

Default setting:

Conversion from array input chaining into column-wise array input is disabled.

Equivalent connection attribute:

SQL_ATTR_COLUMNWISE_MRI

Usage notes:

The multi-row insert (MRI) feature in DB2 for z/OS expects data to be in column-wise array form. Therefore, you can use this conversion to optimize performance in applications that use the array input chaining feature because the data is sent in a compacted form. The following example shows how to enable this conversion:

```
[dsn-name]
...
ColumnwiseMRI=ON
```

The conversion is not performed in certain cases. For details about these cases, see SQL_ATTR_COLUMNWISE_MRI.

This keyword only affects DB2 for z/OS servers.

CommitOnEOF CLI/ODBC configuration keyword

Specifies whether an implicit COMMIT is issued immediately after receiving the last row from a result set. You can free resources as soon as the application receives the entire result set from a cursor by using this keyword.

db2cli.ini keyword syntax:

CommitOnEOF = **0** | **1**

Default setting:

0

Equivalent connection attribute:

SQL_ATTR_COMMITONEOF

Usage notes:

You must enable autocommit and the cursor must be read-only and forward-only to be able to take advantage of this optimization.

If stored procedures or applications return multiple result sets, a COMMIT is issued when the last row from the result set of the last cursor is read.

ConcurrentAccessResolution CLI/ODBC configuration keyword

Specifies the concurrent access resolution to use.

db2cli.ini keyword syntax:

ConcurrentAccessResolution = **0** | **1** | **2** | **3**

Default setting:
DB2 CLI does not supply a prepare option, and the currently committed behavior is determined by the database configuration.

Applicable when:
Connected to a database using DB2 Connect or DB2 Database for Linux, UNIX, and Windows

Equivalent environment or connection attribute:
SQL_ATTR_CONCURRENT_ACCESS_RESOLUTION

Usage notes:
This keyword specifies a prepare attribute that overrides the default behavior specified for cursor stability (CS) scans.

- 0 = No setting. The client does not supply a prepare option.
- 1 = Use currently committed semantics. CLI flows "currently committed" on every prepare, which means that the database manager can use the currently committed version of the data for applicable scans when the data is in the process of being updated or deleted. Rows in the process of being inserted can be skipped. This setting applies when the isolation level in effect is Cursor Stability or Read Stability (for Read Stability it skips uncommitted inserts only) and is ignored otherwise. Applicable scans include read-only scans that can be part of a read-only statement as well as a non read-only statement. The settings for the registry variables **DB2_EVALUNCOMMITTED**, **DB2_SKIPDELETED**, and **DB2_SKIPINSERTED** do not apply to scans using currently committed. However, the settings for these registry variables still apply to scans that do not use currently committed.
- 2 = Wait for outcome. CLI flows "wait for outcome" on every prepare, which means that Cursor Stability and higher scans wait for the commit or rollback when encountering data in the process of being updated or deleted. Rows in the process of being inserted are not skipped. The settings for the registry variables DB2_EVALUNCOMMITTED, DB2_SKIPDELETED, and DB2_SKIPINSERTED no longer apply.
- 3 = Skip locked data. CLI flows "skip locked data" on every prepare, which means that currently committed semantics are used and rows in the process of being inserted are skipped. This option is not supported on DB2 Database for Linux, UNIX, and Windows. If specified, this setting is ignored.

For DB2 Database for Linux, UNIX, and Windows, use this keyword to override the default behavior for currently committed that is defined by the **cur_commit** configuration parameter. For DB2 for z/OS, use this keyword to enable currently committed behavior. There is no equivalent database configuration parameter available on DB2 for z/OS for specifying this behavior.

DB2 z/OS Version 10 only supports INSERT and DELETE operations of currently committed.

ConnectNode CLI/ODBC configuration keyword

Specifies the database partition server to which a connection is to be made.

db2cli.ini keyword syntax:
ConnectNode = **integer value from 0 to 999** |
SQL_CONN_CATALOG_NODE

Default setting:
Database partition server which is defined with port 0 on the machine is used.

Only applicable when:
Connecting to a partitioned database environment.

Equivalent connection attribute:
SQL_ATTR_CONNECT_NODE

Usage notes:

Used to specify the target database partition server that you want to connect to. Can be set to:

- an integer between 0 and 999
- SQL_CONN_CATALOG_NODE

If this variable is not set, the target defaults to the database partition server that is defined with port 0 on the machine.

This keyword (or attribute setting) overrides the value of the **DB2NODE** environment variable. Any out of range value specified for this keyword is ignored and the SQL_CONN_CATALOG_NODE value is used instead.

ConnectTimeout CLI/ODBC configuration keyword

Specifies the time in seconds to wait for a reply when trying to establish a connection to a server before terminating the attempt and generating a communication timeout.

db2cli.ini keyword syntax:
ConnectTimeout = **0** | **1** | **2** | **...** | **32767**

Default setting:
The client waits indefinitely for a reply from the server when trying to establish a connection.

Equivalent connection attribute:
SQL_ATTR_LOGIN_TIMEOUT

Usage notes:

If **ConnectTimeout** is set and client reroute is enabled, a connection will be attempted only once to the original server and once to the alternate server. Since the **ConnectTimeout** value is used when attempting to connect to each server, the maximum waiting time will be approximately double the specified value for **ConnectTimeout**. If neither server can be reached within the amount of time specified by the keyword, the following error message will be received:

```
SQL30081N  A communication error has been detected.  Communication
   protocol being used: "TCP/IP". Communication API being used:
   "SOCKETS".  Location where the error was detected: "<ip address>".
   Communication function detecting the error: "<failing function>".
   Protocol specific error code(s): "<error code>", "*", "*".
   SQLSTATE=08001
```

If **ConnectTimeout** is set and Sysplex exploitation is enabled, a connection will be attempted only once for each of the Sysplex members. Since the **ConnectTimeout** value is used when attempting to connect to each Sysplex member, the maximum waiting time will be approximately equal to the number of Sysplex members, times the amount of time specified by the **ConnectTimeout** keyword.

ConnectTimeout only applies to the TCPIP protocol and is not supported for connections to databases cataloged on a SOCKS-enabled TCP/IP node.

A **ConnectTimeout** value explicitly specified in `db2cli.ini` file will take precedence over the SQL_ATTR_LOGIN_TIMEOUT during the execution.

ConnectType CLI/ODBC configuration keyword

Controls whether the application is to operate in a remote or distributed unit of work.

db2cli.ini keyword syntax:
ConnectType = **1** | **2**

Default setting:
Remote unit of work.

Equivalent environment or connection attribute:
SQL_ATTR_CONNECTTYPE

Usage notes:
This option allows you to specify the default connect type. The options are:

- 1 = Remote unit of work. Multiple concurrent connections, each with its own commit scope. The concurrent transactions are not coordinated. This is the default.
- 2= Distributed unit of work. Coordinated connections where multiple databases participate under the same distributed unit of work.

The first connection determines the connect type for all other connections that are allocated under the same environment handle.

This keyword takes precedence over the environment or connection attribute.

CurrentFunctionPath CLI/ODBC configuration keyword

Specifies the schema used to resolve function references and data type references in dynamic SQL statements.

db2cli.ini keyword syntax:
CurrentFunctionPath = *current_function_path*

Default setting:
See following description.

Usage notes:

This keyword defines the path used to resolve function references and data type references that are used in dynamic SQL statements. It contains a list of one or more schema-names, where schema-names are enclosed in double quotation marks and separated by commas.

The default value is "SYSIBM","SYSFUN",*X* where X is the value of the USER special register delimited by double quotation marks. The schema SYSIBM does not need to be specified. If it is not included in the function path, then it is implicitly assumed as the first schema.

This keyword is used as part of the process for resolving unqualified function and stored procedure references that may have been defined in a schema name other

than the current user's schema. The order of the schema names determines the order in which the function and procedure names will be resolved.

CurrentImplicitXMLParseOption CLI/ODBC configuration keyword

Sets the value of the CURRENT IMPLICIT XMLPARSE OPTION special register.

db2cli.ini keyword syntax:

CurrentImplicitXMLParseOption = **'STRIP WHITESPACE'** | **'PRESERVE WHITESPACE'**

Default setting:

Whitespace is stripped during implicit non-validating parsing.

Equivalent connection attribute:

SQL_ATTR_CURRENT_IMPLICIT_XMLPARSE_OPTION

Usage notes:

Setting this keyword issues the SET CURRENT IMPLICIT XMLPARSE OPTION statement after every connection to a database. By default, this statement is not issued.

The SET CURRENT IMPLICIT XMLPARSE OPTION statement sets the CURRENT IMPLICIT XMLPARSE OPTION special register, which controls whether white space is stripped or preserved during implicit non-validating parsing.

CurrentImplicitXMLParseOption does not affect explicit parsing with the XMLPARSE function.

The supported settings for CurrentImplicitXMLParseOption are:

- STRIP WHITESPACE - white space is removed when an XML document is implicitly parsed. This is the default setting.
- PRESERVE WHITESPACE - white space is preserved when an XML document is implicitly parsed.

CurrentMaintainedTableTypesForOpt CLI/ODBC configuration keyword

Sets the value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register.

db2cli.ini keyword syntax:

CurrentMaintainedTableTypesForOpt = **ALL** | **FEDERATED_TOOL** | **NONE** | **SYSTEM** | **USER** | **<list>**

Default setting:

System-maintained refresh-deferred materialized query tables are considered in the optimization of a query.

Usage notes:

This keyword defines the default value for the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register. The value of the special register affects the types of tables which are considered in the optimization of a query. Refer to the SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION SQL statement for details on the supported settings of ALL, FEDERATED_TOOL, NONE, SYSTEM, or USER. The <*list*> option represents a combination of the

supported settings, however, ALL and NONE cannot be specified with any other value, and the same value cannot be specified more than once. Separate each value in the list with a comma, for example:

```
CurrentMaintainedTableTypesForOpt = SYSTEM,USER
```

CURRENTOPTIMIZATIONPROFILE CLI/ODBC configuration keyword

Specifies the optimization profile used in a SET CURRENT OPTIMIZATION PROFILE statement upon a successful connection.

db2cli.ini keyword syntax:

CURRENTOPTIMIZATIONPROFILE =NULL|*optimization-profile-name*

Default setting:

NULL

Usage notes:

NULL
Sets the register to the null value.

optimization-profile-name
Sets the CURRENT OPTIMIZATION PROFILE special register to the name of an optimization profile. If optimization-profile-name is unqualified, then the default schema qualification is applied.

Examples

If a DB2CLI.INI file has the following entry, after each successful connection to the "Rochester" database, the CLI client would issue the command SET CURRENT OPTIMIZATION PROFILE = '"Hamid"."RochesterProfile"'.

```
[Rochester]
CURRENTOPTIMIZATIONPROFILE='"Hamid"."RochesterProfile"'
```

In this example, the optimization profile name is delimited by quotation marks because it contains lowercase characters.

CurrentPackagePath CLI/ODBC configuration keyword

Issues 'SET CURRENT PACKAGE PATH = *schema1, schema2, ...*' after every connection.

db2cli.ini keyword syntax:

CurrentPackagePath = *schema1, schema2, ...*

Default setting:

The clause is not appended.

Equivalent connection attribute:

SQL_ATTR_CURRENT_PACKAGE_PATH

Usage notes:

When set, this option issues the command "SET CURRENT PACKAGE PATH = *schema1, schema2, ...*" after every connection to the database. This setting specifies the list of schema names (collection identifiers) that will be searched when there is a package from a different schema.

This keyword is best suited for use with ODBC static processing applications, rather than CLI applications.

CurrentPackageSet CLI/ODBC configuration keyword

Issues the SET CURRENT PACKAGESET statement after every connection.

db2cli.ini keyword syntax:
CurrentPackageSet = *schema name*

Default setting:
The clause is not appended.

Equivalent connection attribute:
SQL_ATTR_CURRENT_PACKAGE_SET

Usage notes:

This option issues the SET CURRENT PACKAGESET SQL statement with the CurrentPackageSet value after every connection to a database. By default this clause is not appended.

The SET CURRENT PACKAGESET SQL statement sets the schema name (collection identifier) that is used to select the package to use for subsequent SQL statements.

CLI/ODBC applications issue dynamic SQL statements. Using this option you can control the privileges used to run these statements:

- Choose a schema to use when running SQL statements from CLI/ODBC applications.
- Ensure the objects in the schema have the required privileges and then rebind accordingly.
- Set the CurrentPackageSet option to this schema.

The SQL statements from the CLI/ODBC applications will now run under the specified schema and use the privileges defined there.

The following package set names are reserved: NULLID, NULLIDR1, NULLIDRA.

If both the Reopt and CurrentPackageSet keywords are specified, CurrentPackageSet takes precedence.

CurrentRefreshAge CLI/ODBC configuration keyword

Sets the value of the CURRENT REFRESH AGE special register.

db2cli.ini keyword syntax:
CurrentRefreshAge = **0** | **ANY** | **positive integer**

Default setting:
Only materialized query tables defined with REFRESH IMMEDIATE may be used to optimize the processing of a query.

Usage notes:

Setting this keyword sets the value of the CURRENT REFRESH AGE special register.

CurrentSQLID CLI/ODBC configuration keyword

Specifies the ID used in a SET CURRENT SQLID statement sent to the DBMS upon a successful connection.

db2cli.ini keyword syntax:
CurrentSQLID = *current_sqlid*

Default setting:
No statement is issued.

Only applicable when:
connecting to those DB2 DBMS's where SET CURRENT SQLID is supported.

Usage notes:

Upon a successful connection, if this option is set, a SET CURRENT SQLID statement is sent to the DBMS. This allows the end user and the application to name SQL objects without having to qualify them by schema name.

CurrentSchema CLI/ODBC configuration keyword

Specifies the schema used in a SET CURRENT SCHEMA statement upon a successful connection.

db2cli.ini keyword syntax:
CurrentSchema = *schema name*

Default setting:
No statement is issued.

Usage notes:

Upon a successful connect, if this option is set, a SET CURRENT SCHEMA statement is sent to the DBMS. This allows the end user or application to name SQL objects without having to qualify them by schema name.

CursorHold CLI/ODBC configuration keyword

Controls the effect of a transaction completion on open cursors.

db2cli.ini keyword syntax:
CursorHold = **1** | **0**

Default setting:
Selected--Cursors are not destroyed.

Equivalent statement attribute:
SQL_ATTR_CURSOR_HOLD

Usage notes:

This option controls the effect of a transaction completion on open cursors.
- 1 = SQL_CURSOR_HOLD_ON, the cursors are not destroyed when the transaction is committed (default).
- 0 = SQL_CURSOR_HOLD_OFF, the cursors are destroyed when the transaction is committed.

Note: Cursors are always closed when transactions are rolled back.

This option affects the result returned by `SQLGetInfo()` when called with SQL_CURSOR_COMMIT_BEHAVIOR or SQL_CURSOR_ROLLBACK_BEHAVIOR. The value of CursorHold is ignored if connecting to DB2 Server for VSE & VM where cursor `with hold` is not supported.

You can use this option to tune performance. It can be set to SQL_CURSOR_HOLD_OFF (0) if you are sure that your application:

1. Does not have behavior that is dependent on the SQL_CURSOR_COMMIT_BEHAVIOR or the SQL_CURSOR_ROLLBACK_BEHAVIOR information returned via `SQLGetInfo()`, and
2. Does not require cursors to be preserved from one transaction to the next.

The DBMS will operate more efficiently with CursorHold disabled, as resources no longer need to be maintained after the end of a transaction.

CursorTypes CLI/ODBC configuration keyword

Specifies which cursor types are permitted.

db2cli.ini keyword syntax:

CursorTypes = **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

Default setting:

Forward-only, static, keyset-driven, and dynamic cursors are supported if the server supports them.

Usage notes:

The CursorTypes keyword is a bitmask that indicates what types of cursors an application can open:

- 0x0 - forward-only (can always be opened)
- 0x1 - static
- 0x2 - keyset-driven
- 0x4 - dynamic

For example,

- to prevent applications from opening dynamic scrollable cursors, set CursorTypes to 3.
- to allow applications to open only non-scrollable cursors, set CursorTypes to 0.

This keyword only affects calls made to the following CLI functions:

- `SQLBulkOperations()`
- `SQLExecDirect()`
- `SQLExecute()`
- `SQLFetchScroll()`
- `SQLPrepare()`
- `SQLSetPos()`

DB2Degree CLI/ODBC configuration keyword

Sets the degree of parallelism for the execution of SQL statements.

db2cli.ini keyword syntax:
DB2Degree = **0** | **integer value from 1 to 32767** | **ANY**

Default setting:
No SET CURRENT DEGREE statement is issued.

Only applicable when:
connecting to a cluster database system.

Usage notes:

If the value specified is anything other than 0 (the default) then CLI will issue the following SQL statement after a successful connection:

```
SET CURRENT DEGREE value
```

This specifies the degree of parallelism for the execution of the SQL statements. The database manager will determine the degree of parallelism if you specify ANY.

DB2Explain CLI/ODBC configuration keyword

Determines whether Explain snapshot, Explain table, or both information will be generated by the server.

db2cli.ini keyword syntax:
DB2Explain = **0** | **1** | **2** | **3**

Default setting:
Neither Explain snapshot nor Explain table information will be generated by the server.

Equivalent connection attribute:
SQL_ATTR_DB2EXPLAIN

Usage notes:

This keyword determines whether Explain snapshot, Explain table, or both information will be generated by the server.

- 0 = both off (default)

  A 'SET CURRENT EXPLAIN SNAPSHOT=NO' and a 'SET CURRENT EXPLAIN MODE=NO' statement will be sent to the server to disable both the Explain snapshot and the Explain table information capture facilities.
- 1 = Only Explain snapshot facility on

  A 'SET CURRENT EXPLAIN SNAPSHOT=YES' and a 'SET CURRENT EXPLAIN MODE=NO' statement will be sent to the server to enable the Explain snapshot facility, and disable the Explain table information capture facility.
- 2 = Only Explain table information capture facility on

  A 'SET CURRENT EXPLAIN MODE=YES' and a 'SET CURRENT EXPLAIN SNAPSHOT=NO' will be sent to the server to enable the Explain table information capture facility and disable the Explain snapshot facility.
- 3 = Both on

  A 'SET CURRENT EXPLAIN MODE=YES' and a 'SET CURRENT EXPLAIN SNAPSHOT=YES' will be sent to the server to enable both the Explain snapshot and the Explain table information capture facilities.

Explain information is inserted into Explain tables, which must be created before the Explain information can be generated. The current authorization ID must have INSERT privilege for the Explain tables.

Remarks

Starting in Version 9.7 Fix Pack 3 and later fix packs, DB2 for z/OS server supports only 0 (OFF) and 2 (Table) values in **DB2Explain** settings as only explain mode information is available. If the **DB2Explain** keyword is attempted to set against the data server which do not support it, the application receives an error "CLI0150E Driver not capable".

DB2NETNamedParam CLI/ODBC configuration keyword

Specifies if named parameters are used by DB2 .NET applications.

db2cli.ini keyword syntax:

DB2NETNamedParam = **0** | **1**

Default setting:

The IBM Data Server Provider for .NET recognizes named parameters as parameters, but ignores positioned parameters, in SQL statements.

Usage notes:

By default, theIBM Data Server Provider for .NET processes tokens in an SQL statement with the format "@<paramname>" as named parameters and ignores any positioned parameters, where positioned parameters are specified with a '?' character or a colon followed by a name (*:name*).

The following query is an example of a query that contains a named parameter:

```
SELECT * FROM T1 WHERE C1 = @param1
```

This is an example of a query that contains a positioned parameter:

```
SELECT * FROM T1 WHERE C1 = ?
```

Specify 0 to indicate that only positioned parameters will be recognized as parameters in SQL statements. This setting can improve application performance by reducing the resource required to process named parameters.

DB2Optimization CLI/ODBC configuration keyword

Sets the query optimization level.

db2cli.ini keyword syntax:

DB2Optimization = *integer value from 0 to 9*

Default setting:

No SET CURRENT QUERY OPTIMIZATION statement issued.

Usage notes:

If this option is set then CLI will issue the following SQL statement after a successful connection:

```
SET CURRENT QUERY OPTIMIZATION positive number
```

This specifies the query optimization level at which the optimizer should operate the SQL queries.

DBAlias CLI/ODBC configuration keyword

Specifies the database alias for a data source name (DSN) that is greater than 8 characters.

db2cli.ini keyword syntax:
DBAlias = *dbalias*

Default setting:
Use the DB2 database alias as the ODBC Data Source Name.

Usage notes:

The DSN is the name, enclosed in square brackets, that denotes the section header in the `db2cli.ini` file. Typically, this section header is the database alias name that has a maximum length of 8 bytes. If you want to use a longer, more meaningful name, you can place the longer name in the section header, and set this keyword value to the database alias that is used on the **CATALOG**command. Here is an example:

```
; The much longer name maps to an 8 single byte character dbalias
[MyMeaningfulName]
DBAlias=DB2DBT10
```

You can specify `[MyMeaningfulName]` as the name of the data source on connect while the actual database alias is `DB2DBT10`.

If you specify a value in the **DBAlias** keyword with the database keyword in the `db2cli.ini` file, applications that try to connect to the database or DSN that matches this value do not receive an error.

DBName CLI/ODBC configuration keyword

Specifies the database name to reduce the time it takes for the application to query z/OS or OS/390 table information.

db2cli.ini keyword syntax:
DBName = *dbname*

Default setting:
Do not filter on the DBNAME column.

Only applicable when:
connecting to DB2 for z/OS and OS/390.

Usage notes:

This option is only used when connecting to DB2 for z/OS and OS/390, and only if (*base*) table catalog information is requested by the application. If a large number of tables exist in the z/OS or OS/390 subsystem, a *dbname* can be specified to reduce the time it takes for the application to query table information, and reduce the number of tables listed by the application.

If this option is set then the statement IN DATABASE *dbname* will be appended to various statements such as CREATE TABLE.

This value maps to the DBNAME column in the z/OS or OS/390 system catalog tables. If no value is specified, or if views, synonyms, system tables, or aliases are also specified via TableType, only table information will be restricted; views, aliases, and synonyms are not restricted with DBName. It can be used in conjunction with SchemaList, and TableType to further limit the number of tables for which information will be returned.

DSN CLI/ODBC configuration keyword

Sets the name of a data source as returned by SQLDataSources or the data sources dialog box of SQLDriverConnect.

db2cli.ini keyword syntax:

You can not set this keyword in the db2cli.ini file.

You can specify the value of this keyword in the connection string in SQLDriverConnect like this:

```
DSN = database name
```

Database CLI/ODBC configuration keyword

Specifies the database on the server to connect to when using a File DSN.

db2cli.ini keyword syntax:

Database = *database name*

Default setting:

None

Only applicable when:

Protocol set to TCPIP

Usage notes:

When using a File DSN you must use this option to specify the database on the server to connect to. This value has nothing to do with any database alias name specified on the client, it must be set to the database name on the server itself.

This setting is only considered when the Protocol option is set to TCPIP.

DateTimeStringFormat CLI/ODBC configuration keyword

Specifies the format to use when inserting date or time data into character columns.

db2cli.ini keyword syntax:

DateTimeStringFormat = **JIS** | **ISO** | **EUR** | **USA**

Default setting:

The JIS format is used when date or time data is inserted into character columns.

Usage notes:

The DateTimeStringFormat keyword controls the format in which date or time data is inserted into character columns. This setting affects the insertion of SQL_C_TYPE_DATE, SQL_C_TYPE_TIME, or SQL_C_TYPE_TIMESTAMP, or SQL_C_TIMESTAMP_EXT data into the following column types:

- SQL_CHAR
- SQL_VARCHAR
- SQL_LONGVARCHAR
- SQL_CLOB

This keyword also affects the format of date or time columns that are retrieved into character strings. For example, retrieving data from an SQL_TYPE_TIMESTAMP column into an SQL_C_CHAR string will be affected by the setting of this keyword.

Table 159. Setting Values

Format	Date	Time	Timestamp
JIS	yyyy-mm-dd	hh:mm:ss	yyyy-mm-dd hh:mm:ss.fffffffffff
ISO	yyyy-mm-dd	hh.mm.ss	yyyy-mm-dd-hh.mm.ss.fffffffffff
EUR	dd.mm.yyyy	hh.mm.ss	yyyy-mm-dd hh:mm:ss.fffffffffff*
USA	mm/dd/yyyy	hh:mm AM or PM	yyyy-mm-dd hh:mm:ss.fffffffffff*
*Timestamps will take the default format if EUR or USA is specified. The default format is JIS.			

DecimalFloatRoundingMode CLI/ODBC configuration keyword

Sets the rounding mode when working with servers that support the DECFLOAT SQL type.

db2cli.ini keyword syntax:

DecimalFloatRoundingMode = **0** | 1 | 2 | 3 | 4

Default setting:

0 (Half even rounding mode)

Equivalent connection attribute:

SQL_ATTR_DECFLOAT_ROUNDING_MODE

Usage notes:

The decimal float rounding mode determines what type of rounding will be used if a value is put into a DECFLOAT variable or column but the value has more digits than are allowed in the DECFLOAT data type. This can occur when inserting, updating, selecting, converting from another type, or as the result of a mathematical operation.

The value of SQL_ATTR_DECFLOAT_ROUNDING_MODE determines the decimal float rounding mode that will be used for new connections unless another mode is specified by a connection attribute for that connection. For any given connection both CLI and DB2 will use the same decimal float rounding mode for all action initiated as part of that connection.

When your applications are connecting to a DB2 Database for Linux, UNIX, and Windows Version 9.5 server, you must set the decimal float rounding mode on the database client to the same mode that is set on the server. If you set the decimal float rounding mode on the client to a value that is different from the decimal float rounding mode that is set on the database server, the database server will return SQL0713N on connection.

The settings correspond to these decimal float rounding modes:

- 0 = Half even (default)
- 1 = Half up
- 2 = Down
- 3 = Ceiling
- 4 = Floor

The different modes are:

Half even (default)
In this mode CLI and DB2 use the number that will fit in the target variable and that is closest to the original value. If two numbers are equally close, they use the one that is even. This mode produces the smallest rounding errors over large amounts of data.

Half up
In this mode CLI and DB2 use the number that will fit in the target variable and that is closest to the original value. If two numbers are equally close, they use the one that is greater than the original value.

Down In this mode CLI and DB2 use the number that will fit in the target variable and that is closest to the original value and for which the absolute value is not greater than the absolute value of the original value. You can also think of this as rounding toward zero or as using ceiling for negative values and using floor for positive values.

Ceiling
In this mode CLI and DB2 use the smallest number that will fit in the target variable and that is greater than or equal to the original value.

Floor In this mode CLI and DB2 use the largest number that will fit in the target variable and that is less than or equal to the original value.

This attribute is not supported when accessing IDS data servers.

DeferredPrepare CLI/ODBC configuration keyword

Minimizes network flow by combining the PREPARE request with the corresponding execute request.

db2cli.ini keyword syntax:
DeferredPrepare = **0** | **1**

Default setting:
The prepare request will be delayed until the execute request is sent.

Equivalent statement attribute:
SQL_ATTR_DEFERRED_PREPARE

Usage notes:

Defers sending the PREPARE request until the corresponding execute request is issued. The two requests are then combined into one command/reply flow (instead of two) to minimize network flow and to improve performance.

- 0 = SQL_DEFERRED_PREPARE_OFF. The PREPARE request will be executed the moment it is issued.
- 1 = SQL_DEFERRED_PREPARE_ON (default). Defer the execution of the PREPARE request until the corresponding execute request is issued.

  If the target DBMS does not support deferred prepare, the client disables deferred prepare for that connection.

Note: When deferred prepare is enabled, the row and cost estimates normally returned in the SQLERRD(3) and SQLERRD(4) of the SQLCA of a PREPARE statement may become zeros. This may be of concern to users who want to use these values to decide whether or not to continue the SQL statement.

DescribeCall CLI/ODBC configuration keyword

Determines when stored procedure arguments are described.

db2cli.ini keyword syntax:

DescribeCall = **1** | **-1**

Default setting:

DB2 CLI does not request stored procedure argument describe information when it prepares a CALL statement.

Equivalent connection attribute:

SQL_ATTR_DESCRIBE_CALL

Usage notes:

By default, CLI does not request input parameter describe information when it prepares a CALL statement. If an application has correctly bound parameters to a statement, then this describe information is unnecessary and not requesting it improves performance.

The option values are:

- 1 = SQL_DESCRIBE_CALL_BEFORE. CLI always requests describe information from the server, ignoring the binding information provided by the application. Setting DescribeCall to 1 will also set DeferredPrepare to 0 which means that describe information will also be requested for dynamic SQL statements. Note that setting DeferredPrepare to 0 will not set DescribeCall to 1.
- -1 = SQL_DESCRIBE_CALL_DEFAULT (default). CLI does not request describe information from the server and uses the binding information provided by the application. If the CALL statement execution fails, then the CLI error recovery logic requests input parameter describe information from the server and issues the CALL statement again.

DescribeInputOnPrepare CLI/ODBC configuration keyword

Enables or disables the request for describe information when an SQL statement is prepared.

db2cli.ini keyword syntax:

DescribeInputOnPrepare = **0** | **1**

Default setting:

Do not request describe information when preparing an SQL statement.

Usage notes:

By default, CLI does not request input parameter describe information when it prepares an SQL statement. If an application has correctly bound parameters to a statement, then this describe information is unnecessary and not requesting it improves performance. If, however, parameters have not been correctly bound, then statement execution will fail and cause the CLI error recovery logic to request input parameter describe information. The result is an additional server request and reduced performance, compared to if the describe information had been requested with the prepare. Setting DescribeInputOnPrepare to 1 causes the input describe information to be requested with the prepare. This setting may improve performance for applications which rely heavily on the CLI retry logic to recover from application binding errors.

DescribeOutputLevel CLI/ODBC configuration keyword

Sets the level of output column describe information that is requested by the CLI driver during prepare or describe requests.

db2cli.ini keyword syntax:

DescribeOutputLevel = **0** | **1** | **2** | **3**

Default setting:

Request the describe information listed in level 2 of Table 160 on page 358.

Equivalent connection attribute:

SQL_ATTR_DESCRIBE_OUTPUT_LEVEL

Usage notes:

This keyword controls the amount of information the CLI driver requests on a prepare or describe request. By default, when the server receives a describe request, it returns the information contained in level 2 of Table 160 on page 358 for the result set columns. An application, however, might not need all of this information or might need additional information. Setting the **DescribeOutputLevel** keyword to a level that suits the needs of the client application might improve performance because the describe data transferred between the client and server is limited to the minimum amount that the application requires. If the **DescribeOutputLevel** setting is set too low, it might impact the functionality of the application (depending on the application's requirements). The CLI functions to retrieve the describe information might not fail in this case, but the information returned might be incomplete. Supported settings for **DescribeOutputLevel** are:

- `0` - no describe information is returned to the client application
- `1` - describe information categorized in level 1 (see Table 160 on page 358) is returned to the client application
- `2` - (default) describe information categorized in level 2 (see Table 160 on page 358) is returned to the client application
- `3` - describe information categorized in level 3 (see Table 160 on page 358) is returned to the client application

The following table lists the fields that form the describe information that the server returns when it receives a prepare or describe request. These fields are grouped into levels, and the **DescribeOutputLevel** CLI/ODBC configuration keyword controls which levels of describe information the CLI driver requests.

Note:

1. Not all levels of describe information are supported by all DB2 servers. All levels of describe information are supported on the following DB2 servers: DB2 for Linux, UNIX, and Windows Version 8 and later, DB2 for z/OS Version 8 and later, and DB2 for i5/OS® Version 5 Release 3 and later. All other DB2 servers support only the 2 or 0 setting for **DescribeOutputLevel**.
2. The default behavior allows CLI to promote the level to 3 if the application asks for describe information that was not initially retrieved using the default level 2. This might result in two network flows to the server. If an application uses this keyword to explicitly set a describe level, then no promotion will occur. Therefore, if the keyword is used to set the describe level to 2, CLI will not promote to level 3 even if the application asks for extended information.

Table 160. Levels of describe information

Level 1	Level 2	Level 3
SQL_DESC_COUNT SQL_COLUMN_COUNT SQL_DESC_TYPE SQL_DESC_CONCISE_TYPE SQL_COLUMN_LENGTH SQL_DESC_OCTET_LENGTH SQL_DESC_LENGTH SQL_DESC_PRECISION SQL_COLUMN_PRECISION SQL_DESC_SCALE SQL_COLUMN_SCALE SQL_DESC_DISPLAY_SIZE SQL_DESC_NULLABLE SQL_COLUMN_NULLABLE SQL_DESC_UNSIGNED SQL_DESC_SEARCHABLE SQL_DESC_LITERAL_SUFFIX SQL_DESC_LITERAL_PREFIX SQL_DESC_CASE_SENSITIVE SQL_DESC_FIXED_PREC_SCALE	all fields of level 1 and: SQL_DESC_NAME SQL_DESC_LABEL SQL_COLUMN_NAME SQL_DESC_UNNAMED SQL_DESC_TYPE_NAME SQL_DESC_DISTINCT_TYPE SQL_DESC_REFERENCE_TYPE SQL_DESC_STRUCTURED_TYPE SQL_DESC_USER_TYPE SQL_DESC_LOCAL_TYPE_NAME SQL_DESC_USER_DEFINED_ TYPE_CODE	all fields of levels 1 and 2 and: SQL_DESC_BASE_COLUMN_NAME SQL_DESC_UPDATABLE SQL_DESC_AUTO_UNIQUE_VALUE SQL_DESC_SCHEMA_NAME SQL_DESC_CATALOG_NAME SQL_DESC_TABLE_NAME SQL_DESC_BASE_TABLE_NAME

DescribeParam CLI/ODBC configuration keyword

Enables or disables the SQLDescribeParam() function.

db2cli.ini keyword syntax:

DescribeParam = **0** | **1**

Default setting:

The SQLDescribeParam() function is enabled.

Usage notes:

When set to 1 (default), SQLDescribeParam() is enabled and SQLGetFunctions() will return SQLDescribeParam() as supported.

When set to 0, SQLDescribeParam() is disabled. If SQLDescribeParam() is called, CLI0150E will be returned. SQLGetFunctions() will return SQLDescribeParam() as not supported.

DiagLevel CLI/ODBC configuration keyword

Sets the diagnostic level.

db2cli.ini keyword syntax:
DiagLevel = **0** | **1** | **2** | **3** | **4**

Default setting:
3

Usage notes:
This can be set in the [COMMON] section of the db2cli.ini file only.

This is applicable only at Environment Handle allocation time for an entire process.

This is equivalent to the database manager parameter DIAGLEVEL.

DiagPath CLI/ODBC configuration keyword

Sets the path of the **db2diag** log files.

db2cli.ini keyword syntax:
DiagPath = *existing directory*

Default setting:
The default value is the db2dump directory on UNIX and Linux operating systems, and the db2 directory on Windows operating systems.

Usage notes:

This can be set in the [COMMON] section of the db2cli.ini file only.

This is equivalent to the database manager parameter DIAGPATH.

DisableKeysetCursor CLI/ODBC configuration keyword

Disables keyset-driven scrollable cursors.

db2cli.ini keyword syntax:
DisableKeysetCursor = **0** | **1**

Default setting:
Keyset-driven scrollable cursors are returned when requested.

Usage notes:

When set to 1, this keyword forces the CLI driver to return a static cursor to the application, even if the application has requested a keyset-driven scrollable cursor. The default setting (0) causes keyset-driven cursors to be returned when the application requests them. This keyword can be used to restore behavior before scrollable cursors were supported.

DisableMultiThread CLI/ODBC configuration keyword

Disables multithreading.

db2cli.ini keyword syntax:
DisableMultiThread = **0** | **1**

Default setting:
Multithreading is enabled.

Usage notes:

The CLI/ODBC driver is capable of supporting multiple concurrent threads.

This option is used to enable or disable multi-thread support.
- 0 = Multithreading is enabled (default).
- 1 = Disable multithreading.

If multithreading is disabled then all calls for all threads will be serialized at the process level. Use this setting for multithreaded applications that require serialized behavior.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

DisableUnicode CLI/ODBC configuration keyword

Disables underlying Unicode support.

db2cli.ini keyword syntax:

DisableUnicode = **<not set>** | **0** | **1**

Default setting:

Unicode support is enabled.

Usage notes:

With Unicode support enabled, and when called by a Unicode application, CLI will attempt to connect to the database using the best client code page possible to ensure there is no unnecessary data loss due to code page conversion. This may increase the connection time as code pages are exchanged, or may cause code page conversions on the client that did not occur before this support was added.

If an application is Unicode (the SQL_ATTR_ANSI_APP connection attribute is set to SQL_AA_FALSE, or the connection occurred with SQLConnectW()), then the **DisableUnicode** keyword can be used to effect three different connection behaviors:

- **DisableUnicode** is not set in the db2cli.ini file: If the target database supports Unicode, CLI will connect in Unicode code pages (1208 and 1200). Otherwise, CLI will connect in the application code page.
- **DisableUnicode**=0 is set: CLI always connects in Unicode, whether or not the target database supports Unicode.
- **DisableUnicode**=1 is set: CLI always connects in the application code page, whether or not the target database supports Unicode.

EnableNamedParameterSupport CLI/ODBC configuration keyword

Specifies whether named parameter processing is enabled.

db2cli.ini keyword syntax:

EnableNamedParameterSupport = **TRUE** | **FALSE**

Default setting:

Named parameter support is off (FALSE).

Equivalent environment or connection attribute:

None

Usage notes:

Named parameter processing allows applications to use named parameters (for example, `:name`) in addition to traditional unnamed parameter markers that are represented by a question mark (?). The following options are available to enable or disable named parameter support:

- TRUE - Named parameter processing is enabled. For DB2for Linux, UNIX, and WindowsVersion 9.7 and later, CLI sends the statement text as it is to the server for processing. For all other servers, CLI substitutes the statement text by replacing named parameters with question marks (?) before sending the statement to the server for processing.
- FALSE - Named parameter processing is off, and CLI does not process the parameter markers.

There is no support for the ability to bind by name. CLI processes anything that matches a valid parameter marker, and treats it as if it is a normal parameter marker represented by a question mark (?).

FET_BUF_SIZE CLI/ODBC configuration keyword

Specifies the default query block size to optimize the data flow.

db2cli.ini keyword syntax:

FET_BUF_SIZE = **64K** | **96K** | **128K** | **160K** | **192K** | **224K** | **256K**

Default setting:

FET_BUF_SIZE = 64K

Equivalent connection attribute:

SQL_ATTR_FET_BUF_SIZE

Usage notes:

CLI allows query block size only in multiples of 32K (that is 64K, 96K, 128K, 160K, 192K, 224K, and 256K). CLI applications round up any other values in the range of 64K-256K, to the next nearest 32K boundary.

Application can obtain the value it has set for this attribute using `SQLGetConnectAttr()`. If application has not set any value, the default query block size is returned.

FileDSN CLI/ODBC configuration keyword

Specifies a DSN file from which a connection string will be built for the data source.

db2cli.ini keyword syntax:

You can not set this keyword in the `db2cli.ini` file.

You can specify the value of this keyword in the connection string in SQLDriverConnect like this:

```
FileDSN = file name
```

FloatPrecRadix CLI/ODBC configuration keyword

Forces the NUM_PREC_RADIX value of a floating point type to be 2 or 10.

db2cli.ini keyword syntax:

FloatPrecRadix = **2** | **10**

Default setting:
Report the NUM_PREC_RADIX as 2 for floating point types, as they have a base of 2, not 10.

Usage notes:

The NUM_PREC_RADIX value represents a data type's base. Binary numbers, such as floating point numbers, have a base of 2, and integers have a base of 10. An application may expect all values in the COLUMN_SIZE field to represent the maximum number of digits, which assumes a NUM_PREC_RADIX value of 10. However, for floating point numeric types, the NUM_PREC_RADIX is 2, in which case the COLUMN_SIZE will report the number of bits in the data type's representation, rather than the maximum number of digits.

FloatPrecRadix can force the NUM_PREC_RADIX to be reported as 10 for floating point data types, in which case the COLUMN_SIZE will report the maximum number of digits.

The FloatPrecRadix keyword affects `SQLColumns()`, `SQLGetDescField()` (for the SQL_DESC_NUM_PREC_RADIX field), `SQLGetTypeInfo()`, `SQLProcedureColumns()`, and `SQLSpecialColumns()`.

GetDataLobNoTotal CLI/ODBC configuration keyword

Causes `SQLGetData()` to fetch column data in pieces of specified size (in bytes) instead of fetching column data all at once.

db2cli.ini keyword syntax:
`GetDataLobNoTotal` = *positive integer*

Usage notes:

`SQLGetData()` retrieves data for a single column in the current row of the result set. The first call to `SQLGetData()` results in following tasks:

- Fetches all the data from the database server to the client, which requires allocating memory on the client for the data
- Applies a code page conversion to that data, if required
- Calculates the total length of the converted data
- Returns the length of the converted data to the client application

When the data is large, allocating memory for the data on the client might fail. You can avoid this potential memory allocation problem, by using the **`GetDataLobNoTotal`** keyword.

When you set the **`GetDataLobNoTotal`** keyword, `SQLGetData()` does not fetch all the data for the given column on the first call. Instead, `SQLGetData()` fetches enough data to fill the buffer on the client, as specified by the value of **`GetDataLobNoTotal`**, and returns SQL_NO_TOTAL (-4) if there is more data to be fetched from the server. You can call SQLGetData() as many times as needed to fetch all the data. When all the data has been fetched, SQLGetData() returns SQL_SUCCESS and the size of the last data chunk.

GranteeList CLI/ODBC configuration keyword

Reduces the amount of information returned when the application gets a list of table or column privileges.

db2cli.ini keyword syntax:
GranteeList = " *'userID1', 'userID2',... 'userIDn'* "

Default setting:
Do not filter the results.

Usage notes:

This option can be used to reduce the amount of information returned when the application gets a list of privileges for tables in a database, or columns in a table. The list of authorization IDs specified is used as a filter; the only tables or columns that are returned are those with privileges that have been granted *TO* those IDs.

Set this option to a list of one or more authorization IDs that have been granted privileges, delimited with single quotation mark, and separated by commas. The entire string must also be enclosed in double quotation marks. For example:

```
GranteeList=" 'USER1', 'USER2', 'USER8' "
```

In the example, if the application gets a list of privileges for a specific table, only those columns that have a privilege granted *TO* USER1, USER2, or USER8 would be returned.

GrantorList CLI/ODBC configuration keyword

Reduces the amount of information returned when the application gets a list of table or column privileges.

db2cli.ini keyword syntax:
GrantorList = " *'userID1', 'userID2',... 'userIDn'* "

Default setting:
Do not filter the results.

Usage notes:

This option can be used to reduce the amount of information returned when the application gets a list of privileges for tables in a database, or columns in a table. The list of authorization IDs specified is used as a filter; the only tables or columns that are returned are those with privileges that have been granted *BY* those IDs.

Set this option to a list of one or more authorization IDs that have granted privileges, delimited with single quotation mark, and separated by commas. The entire string must also be enclosed in double quotation marks. For example:

```
GrantorList=" 'USER1', 'USER2', 'USER8' "
```

In the example, if the application gets a list of privileges for a specific table, only those columns that have a privilege granted *BY* USER1, USER2, or USER8 would be returned.

Graphic CLI/ODBC configuration keyword

Specifies if CLI returns SQL_GRAPHIC (double-byte character) as a supported SQL data type and what unit is used to report GRAPHIC column length.

db2cli.ini keyword syntax:
Graphic = **0** | **1** | **2** | **3**

Default setting:
The SQL_GRAPHIC data type is not returned as a supported SQL data type, and the length of GRAPHIC columns equals the maximum number of DBCS characters in the column.

Usage Notes:

The Graphic keyword controls whether the SQL_GRAPHIC (double-byte character) data type is reported as a supported SQL data type when `SQLGetTypeInfo()` is called, as well as what unit is used to report the length of GRAPHIC columns for all CLI functions that return length or precision as either output arguments or as part of a result set.

Set the Graphic keyword as follows:

- 0 - SQL_GRAPHIC is not returned as a supported SQL data type, and the reported length of GRAPHIC columns equals the maximum number of DBCS characters in the column.
- 1 - SQL_GRAPHIC is returned as a supported SQL data type, and the reported length of GRAPHIC columns equals the maximum number of DBCS characters in the column.
- 2 - SQL_GRAPHIC is not returned as a supported SQL data type, and the reported length of GRAPHIC columns equals the maximum number of bytes in the column.
- 3 - SQL_GRAPHIC is returned as a supported SQL data type, and the reported length of GRAPHIC columns equals the maximum number of bytes in the column.

Hostname CLI/ODBC configuration keyword

Specifies the server system's host name or IP address, used with file DSN or in a DSN-less connection.

db2cli.ini keyword syntax:
Hostname = *host name* | *IP Address*

Default setting:
None

Only applicable when:
Protocol set to TCPIP

Usage notes:

Use this option in conjunction with the ServiceName option to specify the required attributes for a TCP/IP connection from this client machine to a server running DB2. These two values are only considered when the Protocol option is set to TCPIP.

Specify either the server system's host name or its IP address.

IgnoreWarnList CLI/ODBC configuration keyword

Ignores specified sqlstates.

db2cli.ini keyword syntax:

IgnoreWarnList = "'sqlstate1', 'sqlstate2', ..."

Default setting:

Warnings are returned as normal

Usage notes:

On rare occasions an application may not correctly handle some warning messages, but does not want to ignore all warning messages. This keyword can be used to indicate which warnings are not to be passed on to the application. The IgnoreWarnings keyword should be used if all database manager warnings are to be ignored.

If an sqlstate is included in both IgnoreWarnList and WarningList, it will be ignored altogether.

Each sqlstate must be in uppercase, delimited with single quotation mark and separated by commas. The entire string must also be enclosed in double quotation marks. For example:

```
IgnoreWarnList="'01000', '01004','01504'"
```

IgnoreWarnings CLI/ODBC configuration keyword

Ignores database manager warnings.

db2cli.ini keyword syntax:

IgnoreWarnings = **0** | **1**

Default setting:

Warnings are returned as normal.

Usage notes:

On rare occasions, an application will not correctly handle warning messages. This keyword can be used to indicate that warnings from the database manager are not to be passed to the application. The possible settings are:

- 0 - Warnings are reported as usual (default)
- 1 - Database manager warnings are ignored and SQL_SUCCESS is returned. Warnings from the DB2 CLI/ODBC driver are still returned; many are required for normal operation.

Although this keyword can be used on its own, it can also be used with the WarningList CLI/ODBC configuration keyword.

Instance CLI/ODBC configuration keyword

Specifies the instance name for a local IPC connection for file DSN or DSN-less connectivity.

db2cli.ini keyword syntax:

Instance = *instance name*

Usage notes:

This can be set in the `[Data Source]` section of the `db2cli.ini` file for the given data source, or in a connection string.

When you set this keyword, you must also set the following options:

- Database
- Protocol=IPC

Interrupt CLI/ODBC configuration keyword

Sets the interrupt processing mode.

db2cli.ini keyword syntax:
Interrupt = **0** | **1** | **2**

Default setting:
1

Usage notes:
This can be set in the [Data Source] section of the db2cli.ini file for the given data source, or in a connection string.

When you set this option, you must also set the following options:

- Database
- Protocol=IPC

The keyword values have the following meaning:

0 Disables interrupt processing (SQLCancel calls will not interrupt the processing.)

1 Interrupts are supported (default.) In this mode, if the server supports an interrupt, an interrupt will be sent. Otherwise the connection is dropped.

The settings for INTERRUPT_ENABLED (a DB2 Connect gateway setting) and the DB2 registry variable DB2CONNECT_DISCONNECT_ON_INTERRUPT will take precedence over the Interrupt keyword setting of 1.

2 Interrupt drops the connection regardless of server's interrupt capabilities (SQLCancel will drop the connection.)

KRBPlugin CLI/ODBC configuration keyword

Specifies the name of the Kerberos plug-in library to be used for client side authentication for file DSN or DSN-less connectivity.

db2cli.ini keyword syntax:
KRBPlugin = *plugin name*

Default setting:
By default, the value is null on UNIX operating systems, and IBMkrb5 on Windows operating systems.

Usage notes:
This can be set in the [Data Source] section of the db2cli.ini file for the given data source, or in a connection string.

This parameter specifies the name of the Kerberos plug-in library to be used for client-side connection authentication. The plug-in is used when the client is authenticated using KERBEROS authentication.

KeepDynamic CLI/ODBC configuration keyword

Specifies if KEEPDYNAMIC functionality is available to CLI applications.

db2cli.ini keyword syntax:
KeepDynamic = **0** | **1**

Default setting:
KEEPDYNAMIC functionality is not available to CLI applications.

Equivalent connection attribute:
SQL_ATTR_KEEP_DYNAMIC

Usage notes:

The KeepDynamic CLI/ODBC configuration keyword should be set according to how the CLI packages were bound on the DB2 for z/OS and OS/390 server. Set KeepDynamic as follows:

- 0 - if the CLI packages on the server were bound with the KEEPDYNAMIC NO option
- 1 - if the CLI packages on the server were bound with the KEEPDYNAMIC YES option

It is recommended that when KeepDynamic is used, the CurrentPackageSet CLI/ODBC keyword also be set. Refer to the documentation about enabling KEEPDYNAMIC support for details on how these keywords can be used together.

LOBCacheSize CLI/ODBC configuration keyword

Specifies maximum cache size (in bytes) for LOBs.

db2cli.ini keyword syntax:
LOBCacheSize = *positive integer*

Default setting:
LOBs are not cached.

Equivalent connection or statement attribute:
SQL_ATTR_LOB_CACHE_SIZE

Usage notes:

The use of LOB locators when retrieving unbound LOB data can be avoided by setting this keyword. For example, if an application does not bind a column before calling `SQLFetch()` and then calls `SQLGetData()` to fetch the LOB, if LOBCacheSize was set to a value large enough to contain the entire LOB being fetched, then the LOB is retrieved from the LOB cache rather than from a LOB locator. Using the LOB cache instead of the LOB locator in this case improves performance.

Servers that support Dynamic Data Format, also known as progressive streaming, optimize the return of LOB and XML data depending on the actual length of the data. The LOB and XML data can be returned in its entirety, or as an internal token called a progressive reference. CLI manages progressive reference data retrieval.

For applications that are querying data on a server that supports Dynamic Data Format, setting the LOBCacheSize keyword sets a threshold that is used to determine if the data is returned in its entirety, or as a progressive reference. If the data has a length greater than the LOBCacheSize threshold value, the progressive reference will be returned to CLI to manage, but if the data has a length less than or equal to the LOBCacheSize threshold value, the data will be returned in its entirety.

For applications that are querying data on a server that does not supports Dynamic Data Format, the LOBCacheSize threshold value specifies the maximum defined size of a LOB that CLI will buffer in memory. If the defined size of a LOB exceeds the value LOBCacheSize is set to, then the LOB will not be cached. For example, consider a table that is created with a CLOB column of 100MB currently holding 20MB of data, with LOBCacheSize set to 50MB. In this case, even though the size of the LOB itself (20MB) is less than the value set through LOBCacheSize, the CLOB column will not be cached because the defined CLOB size (100MB) exceeds the maximum cache size set through LOBCacheSize (50MB).

ClientBuffersUnboundLOBS is a related keyword.

LOBFileThreshold CLI/ODBC configuration keyword

Specifies the maximum number of bytes of LOB data buffered when `SQLPutData()` is used.

db2cli.ini keyword syntax:
LOBFileThreshold = *positive integer*

Default setting:
25 MB

Usage notes:

This option specifies the maximum number of bytes of LOB data that CLI will buffer in memory on calls to `SQLPutData()`. If the specified cache size is exceeded, a temporary file will be created on disk to hold the LOB data before it is sent to the server.

LOBMaxColumnSize CLI/ODBC configuration keyword

Overrides the default value in the COLUMN_SIZE column for LOB data types.

db2cli.ini keyword syntax:
LOBMaxColumnSize = *integer greater than zero*

Default setting:
2 Gigabytes (1G for DBCLOB)

Only applicable when:
LongDataCompat or MapXMLDescribe with a LOB type is used.

Usage notes:

This will override the 2 Gigabyte (1G for DBCLOB) value that is returned by `SQLGetTypeInfo()` for the COLUMN_SIZE column for SQL_CLOB, SQL_BLOB, and SQL_DBCLOB and SQL_XML SQL data types. For SQL_XML, LOBMaxColumnSize must be specified with MapXMLDescribe set to a LOB type. Subsequent CREATE TABLE statements that contain LOB columns will use the column size value you set here instead of the default.

LoadXAInterceptor CLI/ODBC configuration keyword

Loads the XA Interceptor for debugging.

db2cli.ini keyword syntax:
LoadXAInterceptor = **0** | **1**

Default setting:
The XA Interceptor is not loaded.

Usage notes:

This keyword loads the XA Interceptor for debugging purposes in MTS.

LockTimeout CLI/ODBC configuration keyword

Sets the default value of the LOCKTIMEOUT configuration parameter.

db2cli.ini keyword syntax:
LockTimeout = **-1** | **0** | **positive integer ≤ 32767**

Default setting:
Timeout is turned off (-1), with the application waiting for a lock until either the lock is granted or deadlock occurs.

Usage notes:

The LockTimeout keyword specifies the number of seconds a CLI application will wait to obtain locks. If the keyword is set to 0, locks will not be waited for. The -1 setting causes the application to wait indefinitely until either the lock is granted or deadlock occurs.

LongDataCompat CLI/ODBC configuration keyword

Reports LOBs as long data types or as large object types.

db2cli.ini keyword syntax:
LongDataCompat = **0** | **1**

Default setting:
Reference LOB data types as large object types.

Equivalent connection attribute:
SQL_ATTR_LONGDATA_COMPAT

Usage notes:

This option indicates to CLI what data type the application expects when working with a database with large object (LOB) columns.

The values for this option are:

- 0 = Reference LOB data types as large object types.
- 1 = Report LOBs as long data types for CLI/ODBC applications only.

Table 161. Corresponding large object and long data types for LOB data

Database data type	Large objects (0 - Default)	Long data types (1 — CLI/ODBC)
CLOB	SQL_CLOB	SQL_LONGVARCHAR
BLOB	SQL_BLOB	SQL_LONGVARBINARY
DBCLOB	SQL_DBCLOB	SQL_LONGVARGRAPHIC*

Table 161. Corresponding large object and long data types for LOB data (continued)

Database data type	Large objects (0 - Default)	Long data types (1 — CLI/ODBC)

* If the MapGraphicDescribe keyword is set in conjunction with LongDataCompat, DBCLOB columns will return an SQL type of SQL_LONGVARCHAR if MapGraphicDescribe is 1 and SQL_WLONGVARCHAR if MapGraphicDescribe is 2.

This option is useful when running ODBC applications that cannot handle the large object data types.

The CLI/ODBC option LOBMaxColumnSize can be used in conjunction with this option to reduce the default size declared for the data.

MapBigintCDefault CLI/ODBC configuration keyword

Specifies the default C type of BIGINT columns and parameter markers.

db2cli.ini keyword syntax:

MapBigintCDefault = **0** | **1** | **2**

Default setting:

The default C type representation for BIGINT data is SQL_C_BIGINT.

Usage notes:

MapBigintCDefault controls the C type that is used when SQL_C_DEFAULT is specified for BIGINT columns and parameter markers. This keyword should be used primarily with Microsoft applications, such as Microsoft Access, which cannot handle 8-byte integers. Set MapBigintCDefault as follows:

- 0 - for the default SQL_C_BIGINT C type representation
- 1 - for an SQL_C_CHAR C type representation
- 2 - for an SQL_C_WCHAR C type representation

This keyword affects the behavior of CLI functions where SQL_C_DEFAULT might be specified as a C type, such as `SQLBindParameter()`, `SQLBindCol()`, and `SQLGetData()`

MapCharToWChar CLI/ODBC configuration keyword

Specifies the default SQL type associated with SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR.

db2cli.ini keyword syntax:

MapCharToWChar = **0** | **1**

Default setting:

The default SQL type representation for SQL_CHAR, SQL_VARCHAR and SQL_LONGVARCHAR is used.

Equivalent connection attribute:

SQL_ATTR_MAPCHAR

Usage notes:

MapCharToWChar controls the SQL type that is returned when describing SQL_CHAR, SQL_VARCHAR and SQL_LONGVARCHAR columns or parameter markers.

Set MapCharToWChar as follows:

- 0 - to return the default SQL type representation
- 1 - to return SQL_CHAR as SQL_WCHAR, SQL_VARCHAR as SQL_WVARCHAR, and SQL_LONGVARCHAR as SQL_WLONGVARCHAR

Only the following CLI functions are affected by setting MapCharToWChar:

- SQLColumns()
- SQLColAttribute()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLGetDescField()
- SQLGetDescRec()
- SQLProcedureColumns()

MapDateCDefault CLI/ODBC configuration keyword

Specifies the default C type of DATE columns and parameter markers.

db2cli.ini keyword syntax:

MapDateCDefault = **0** | **1** | **2**

Default setting:

The default C type representation for DATE data is SQL_C_TYPE_DATE.

Usage notes:

MapDateCDefault controls the C type that is used when SQL_C_DEFAULT is specified for DATE columns and parameter markers. This keyword should be used primarily with Microsoft applications, such as Microsoft Access, which assume SQL_C_CHAR as the default C type for datetime values. Set MapDateCDefault as follows:

- 0 - for the default SQL_C_TYPE_DATE C type representation: a struct containing numeric members for year, month and day
- 1 - for an SQL_C_CHAR C type representation: "2004-01-01"
- 2 - for an SQL_C_WCHAR C type representation: "2004-01-01" in UTF-16.

This keyword affects the behavior of CLI functions where SQL_C_DEFAULT may be specified as a C type, such as `SQLBindParameter()`, `SQLBindCol()`, and `SQLGetData()`.

MapDateDescribe CLI/ODBC configuration keyword

Controls the SQL data type returned when DATE columns and parameter markers are described.

db2cli.ini keyword syntax:

MapDateDescribe = **0** | **1** | **2**

Default setting:

The default SQL data type for DATE data is returned: SQL_DATE for ODBC 2.0 or SQL_TYPE_DATE for ODBC 3.0.

Usage notes:

MapDateDescribe CLI/ODBC configuration keyword

To control the SQL data type that is returned when DATE columns and parameter markers are described, set MapDateDescribe as follows:

- 0 - to return the default SQL data type: SQL_DATE for ODBC 2.0 or SQL_TYPE_DATE for ODBC 3.0
- 1 - to return the SQL_CHAR SQL data type
- 2 - to return the SQL_WCHAR SQL data type

Only the following CLI functions are affected by setting MapDateDescribe:

- `SQLColumns()`
- `SQLDescribeCol()`
- `SQLDescribeParam()`
- `SQLGetDescField()`
- `SQLGetDescRec()`
- `SQLProcedureColumns()`
- `SQLSpecialColumns()`

MapDecimalFloatDescribe CLI/ODBC configuration keyword

Specifies the default C type and reported data type of DECFLOAT columns and parameter markers.

db2cli.ini keyword syntax:

MapDecimalFloatDescribe = 0 | 1 | 2 | 3

Default setting:

0

Usage notes:

MapDecimalFloatDescribe controls the default C type to be used for columns and parameters with a data type of DECFLOAT. It affects the behavior of CLI functions for which SQL_C_DEFAULT can be specified as the C type of a column or parameter. Examples of such functions include SQLBindParameter(), SQLBindCol(), and SQLGetData().

MapDecimalFloatDescribe also controls the type that will be reported for columns and parameters that have a data type of DECFLOAT. This affects CLI functions that return information about parameters and columns. Examples of such functions include SQLColAttribute() and SQLDescribeParam().

Use this configuration keyword for applications that cannot handle decimal float types or when you would rather always deal with decimal float types as some other type.

Here are the allowed values:

Table 162. Valid values for MapDecimalFloatDescribe

Value	DECFLOAT columns and parameters are reported as being this type	DECFLOAT columns and parameters use this default C type
0	SQL_DECFLOAT	SQL_C_CHAR
1	SQL_VARCHAR	SQL_C_CHAR
2	SQL_WVARCHAR	SQL_C_WCHAR

Table 162. Valid values for MapDecimalFloatDescribe (continued)

Value	DECFLOAT columns and parameters are reported as being this type	DECFLOAT columns and parameters use this default C type
3	SQL_DOUBLE	SQL_C_DOUBLE

MapGraphicDescribe CLI/ODBC configuration keyword

Controls the SQL data type returned when GRAPHIC, VARGRAPHIC, and LONGVARGRAPHIC columns and parameter markers are described.

db2cli.ini keyword syntax:

MapGraphicDescribe = **0** | **1** | **2**

Default setting:

The default SQL data types are returned: SQL_GRAPHIC for GRAPHIC columns, SQL_VARGRAPHIC for VARGRAPHIC columns, and SQL_LONGVARGRAPHIC for LONG VARGRAPHIC columns.

Usage notes:

To control the SQL data type that is returned when GRAPHIC-based columns and parameter markers are described, set MapGraphicDescribe as follows:

- 0 - to return the default SQL data types
- 1 - to return the CHAR-based SQL data types: SQL_CHAR for GRAPHIC columns, SQL_VARCHAR for VARGRAPHIC columns, and SQL_LONGVARCHAR for LONG VARGRAPHIC columns
- 2 - to return the WCHAR-based SQL data types: SQL_WCHAR for GRAPHIC columns, SQL_WVARCHAR for VARGRAPHIC columns, and SQL_WLONGVARCHAR for LONG VARGRAPHIC columns

Only the following CLI functions are affected by setting MapGraphicDescribe:

- `SQLDescribeCol()`
- `SQLDescribeParam()`
- `SQLGetDescField()`
- `SQLGetDescRec()`
- `SQLProcedureColumns()`
- `SQLSpecialColumns()`

MapTimeCDefault CLI/ODBC configuration keyword

Specifies the default C type of TIME columns and parameter markers.

db2cli.ini keyword syntax:

MapTimeCDefault = **0** | **1** | **2**

Default setting:

The default C type representation for TIME data is SQL_C_TYPE_TIME.

Usage notes:

MapTimeCDefault controls the C type that is used when SQL_C_DEFAULT is specified for TIME columns and parameter markers. This keyword should be used

primarily with Microsoft applications, such as Microsoft Access, which assume SQL_C_CHAR as the default C type for datetime values. Set MapTimeCDefault as follows:

- 0 - for the default SQL_C_TYPE_TIME C type representation: a struct containing numeric members for hour, minute, and second
- 1 - for an SQL_C_CHAR C type representation: "12:34:56"
- 2 - for an SQL_C_WCHAR C type representation: "12:34:56" in UTF-16.

This keyword affects the behavior of CLI functions where SQL_C_DEFAULT may be specified as a C type, such as `SQLBindParameter()`, `SQLBindCol()`, and `SQLGetData()`.

Note: MapTimeCDefault supersedes Patch2=24. If both MapTimeCDefault and Patch2=24 are set, the MapTimeCDefault value takes precedence.

MapTimeDescribe CLI/ODBC configuration keyword

Controls the SQL data type returned when TIME columns and parameter markers are described.

db2cli.ini keyword syntax:

MapTimeDescribe = **0** | **1** | **2**

Default setting:

The default SQL data type for TIME data is returned: SQL_TIME for ODBC 2.0 or SQL_TYPE_TIME for ODBC 3.0

Usage notes:

To control the SQL data type that is returned when TIME columns and parameter markers are described, set MapTimeDescribe as follows:

- 0 - to return the default SQL data type: SQL_TIME for ODBC 2.0 or SQL_TYPE_TIME for ODBC 3.0
- 1 - to return the SQL_CHAR SQL data type
- 2 - to return the SQL_WCHAR SQL data type

Only the following CLI functions are affected by setting MapTimeDescribe:

- `SQLColumns()`
- `SQLDescribeCol()`
- `SQLDescribeParam()`
- `SQLGetDescField()`
- `SQLGetDescRec()`
- `SQLProcedureColumns()`
- `SQLSpecialColumns()`

MapTimestampCDefault CLI/ODBC configuration keyword

Specifies the default C type of TIMESTAMP columns and parameter markers.

db2cli.ini keyword syntax:

MapTimestampCDefault = **0** | **1** | **2**

Default setting:

The default C type representation for TIMESTAMP data is SQL_C_TYPE_TIMESTAMP.

Usage notes:

MapTimestampCDefault controls the C type that is used when SQL_C_DEFAULT is specified for TIMESTAMP columns and parameter markers. This keyword should be used primarily with Microsoft applications, such as Microsoft Access, which assume SQL_C_CHAR as the default C type for datetime values. Set MapTimestampCDefault as follows:

- 0 - for the default SQL_C_TYPE_TIMESTAMP C type representation: a struct containing numeric members for year, month, day, hour, minute, second, and fraction of a second
- 1 - for an SQL_C_CHAR C type representation: "2004-01-01 12:34:56.123...*n*", where *n*=12.
- 2 - for an SQL_C_WCHAR C type representation: "2004-01-01 12:34:56.123456...*n*"" in UTF-16, where *n*=12.

This keyword affects the behavior of CLI functions where SQL_C_DEFAULT may be specified as a C type, such as `SQLBindParameter()`, `SQLBindCol()`, and `SQLGetData()`.

MapTimestampDescribe CLI/ODBC configuration keyword

Controls the SQL data type returned when TIMESTAMP columns and parameter markers are described.

db2cli.ini keyword syntax:

MapTimestampDescribe = **0** | **1** | **2**

Default setting:

The default SQL data type for TIMESTAMP data is returned: SQL_TIMESTAMP for ODBC 2.0 or SQL_TYPE_TIMESTAMP for ODBC 3.0.

Usage notes:

To control the SQL data type that is returned when TIMESTAMP columns and parameter markers are described, set MapTimestampDescribe as follows:

- 0 - to return the default SQL data type: SQL_TIMESTAMP for ODBC 2.0 or SQL_TYPE_TIMESTAMP for ODBC 3.0
- 1 - to return the SQL_CHAR SQL data type
- 2 - to return the SQL_WCHAR SQL data type

Only the following CLI functions are affected by setting MapTimeStampDescribe:

- `SQLColumns()`
- `SQLDescribeCol()`
- `SQLDescribeParam()`
- `SQLGetDescField()`
- `SQLGetDescRec()`
- `SQLProcedureColumns()`
- `SQLSpecialColumns()`

MapXMLCDefault CLI/ODBC configuration keyword

Controls the default C type representation used when SQL_C_DEFAULT is specified for XML columns and parameter markers.

db2cli.ini keyword syntax:

MapXMLCDefault = **0** | **1** | **2** | **3**

Default setting:

The default C type representation for XML data is SQL_C_BINARY.

Usage notes:

MapXMLCDefault controls the C type that is used when SQL_C_DEFAULT is specified for XML columns and parameter markers. This keyword should be used primarily with Microsoft applications, such as Microsoft Access, which might assume SQL_C_WCHAR as the default C type for XML values. Set MapXMLCDefault as follows:

- 0 - for the default SQL_C_BINARY C type representation
- 1 - for the SQL_C_CHAR C type representation; this can result in data loss as the XML data is converted to the local application code page
- 2 - for the SQL_C_WCHAR C type representation

This keyword affects the behaviour of CLI functions where SQL_C_DEFAULT can be specified as a C type, such as `SQLBindParameter()`, `SQLBindCol()`, and `SQLGetData()`.

MapXMLDescribe CLI/ODBC configuration keyword

Controls the SQL data type returned when XML columns and parameter markers are described.

db2cli.ini keyword syntax:

MapXMLDescribe = **-370** | **-350** | **-152** | **-99** | **-98**

Default setting:

The default SQL data type for XML data is returned: SQL_XML (-370)

Usage notes:

To control the SQL data type that is returned when XML columns and parameter markers are described, set MapXMLDescribe to one of the following integer values:

- -370 to return the default SQL_XML SQL data type
- -350 to return the SQL_DBCLOB SQL data type
- -152 to return the SQL_SS_XML SQL data type

  Note: The SQL_SS_XML value of -152 belongs to the reserved range of Microsoft SQL Server and is not defined by IBM.
- -99 to return the SQL_BLOB SQL data type
- -98 to return the SQL_CLOB SQL data type

The data length for XML values mapped to LOB types is the maximum length for the mapped data type.

When used in conjunction with the LongDataCompat keyword set to the value 1, XML values mapped to LOB data types will be mapped to the corresponding LONG data type as well.

Character types specified for MapXMLDescribe may result in data loss during data conversion if the application code page does not support all of the characters in the source data. Mapping XML values to character types, therefore, is only recommended with caution.

This keyword is recommended to provide compatibility with applications that access XML columns as CLOB or BLOB, or use Microsoft application development technologies.

MaxLOBBlockSize CLI/ODBC configuration keyword

Specifies the maximum return block size for LOB or XML data.

db2cli.ini keyword syntax:

MaxLOBBlockSize = **0** | **...** | **2147483647**

Default setting:

There is no limit to the data block size for LOB or XML data.

Equivalent connection or statement attribute:

SQL_ATTR_MAX_LOB_BLOCK_SIZE

Usage notes:

During data retrieval, the server will include all of the information for the current row in its reply to the client even if the maximum block size has been reached.

If both MaxLOBBlockSize and the db2set registry variable DB2_MAX_LOB_BLOCK_SIZE are specified, the value for MaxLOBBlockSize will be used.

Mode CLI/ODBC configuration keyword

Sets the default connection mode.

db2cli.ini keyword syntax:

Mode = **SHARE** | **EXCLUSIVE**

Default setting:

SHARE

Not applicable when:

connecting to a host or IBM Power Systems server.

Usage notes:

Sets the CONNECT mode to either SHARE or EXCLUSIVE. If a mode is set by the application at connect time, this value is ignored. The default is SHARE.

NotifyLevel CLI/ODBC configuration keyword

Sets the diagnostic level.

db2cli.ini keyword syntax:

NotifyLevel = **0** | **1** | **2** | **3** | **4**

Default setting:
3

Usage notes:
This can be set in the [COMMON] section of the db2cli.ini file only.

This is equivalent to the database manager parameter NOTIFYLEVEL.

OleDbReportIsLongForLongTypes CLI/ODBC configuration keyword

Makes OLE DB flag LONG data types with DBCOLUMNFLAGS_ISLONG.

db2cli.ini keyword syntax:
OleDbReportIsLongForLongTypes = **0** | **1**

Equivalent connection attribute:
SQL_ATTR_REPORT_ISLONG_FOR_LONGTYPES_OLEDB

Default setting:
LONG types (LONG VARCHAR, LONG VARCHAR FOR BIT DATA, LONG VARGRAPHIC and LONG VARGRAPHIC FOR BIT DATA) do not have the DBCOLUMNFLAGS_ISLONG flag set, which might cause the columns to be used in the WHERE clause.

Usage notes:
The OLE DB client cursor engine and the OLE DB .NET Data Provider CommandBuilder object generate UPDATE and DELETE statements based on column information provided by the IBM DB2 OLE DB Provider. If the generated statement contains a LONG type in the WHERE clause, the statement will fail because LONG types cannot be used in a search with an equality operator. Setting the keyword OleDbReportIsLongForLongTypes to 1 will make the IBM DB2 OLE DB Provider report LONG types (LONG VARCHAR, LONG VARCHAR FOR BIT DATA, LONG VARGRAPHIC and LONG VARGRAPHIC FOR BIT DATA) with the DBCOLUMNFLAGS_ISLONG flag set. This will prevent the long columns from being used in the WHERE clause.

The OleDbReportIsLongForLongTypes keyword is supported by the following database servers:

- DB2 for z/OS
  - version 6 with PTF UQ93891
  - version 7 with PTF UQ93889
  - version 8 with PTF UQ93890
  - versions later than version 8, PTFs are not required
- DB2 Database for Linux, UNIX, and Windows
  - version 8.2 (equivalent to Version 8.1, FixPak 7) and later

OleDbReturnCharAsWChar CLI/ODBC configuration keyword

Controls how the IBM DB2 OLE DB Provider describes CHAR, VARCHAR, LONG VARCHAR, and CLOB data.

db2cli.ini keyword syntax:
OleDbReturnCharAsWChar = **0** | **1**

Default setting:
The IBM DB2 OLE DB Provider describes CHAR, VARCHAR, LONG VARCHAR, and CLOB data as DBTYPE_WSTR.

Usage notes:

The IBM DB2 OLE DB Provider describes CHAR, VARCHAR, LONG VARCHAR, and CLOB data as DBTYPE_WSTR by default as of DB2 luw Version 8.1.2. The CLI/ODBC configuration keyword OleDbReturnCharAsWChar allows you to change this default to have the previously stated character data types reported as DBTYPE_STR.

The available settings are:

- 0 - CHAR, VARCHAR, LONG VARCHAR, and CLOB data are described as DBTYPE_STR, and the code page of data in ISequentialStream is the local code page of the client
- 1 - CHAR, VARCHAR, LONG VARCHAR, and CLOB data are reported as DBTYPE_WSTR, and the code page of data in ISequentialStream is UCS-2

OleDbSQLColumnsSortByOrdinal CLI/ODBC configuration keyword

Makes OLE DB's IDBSchemaRowset::GetRowset(DBSCHEMA_COLUMNS) return a row set sorted by the ORDINAL_POSITION column.

db2cli.ini keyword syntax:

OleDbSQLColumnsSortByOrdinal = **0** | **1**

Equivalent connection attribute:

SQL_ATTR_SQLCOLUMNS_SORT_BY_ORDINAL_OLEDB

Default setting:

IDBSchemaRowset::GetRowset(DBSCHEMA_COLUMNS) returns the row set sorted by the columns TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME.

Usage notes:

The Microsoft OLE DB specification requires that IDBSchemaRowset::GetRowset(DBSCHEMA_COLUMNS) returns the row set sorted by the columns TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME. The IBM DB2 OLE DB Provider conforms to the specification. However, applications that use the Microsoft ODBC Bridge provider (MSDASQL) have been typically coded to get the row set sorted by ORDINAL_POSITION. Setting the OleDbSQLColumnsSortByOrdinal keyword to 1 will make the provider return a row set sorted by ORDINAL_POSITION.

The OleDbSQLColumnsSortByOrdinal keyword is supported by the following database servers:

- DB2 for z/OS
  - version 6 with PTF UQ93891
  - version 7 with PTF UQ93889
  - version 8 with PTF UQ93890
  - versions later than version 8, PTFs are not required
- DB2 Database for Linux, UNIX, and Windows
  - version 8.2 (equivalent to Version 8.1, FixPak 7) and later

OnlyUseBigPackages CLI/ODBC configuration keyword

Allows the CLI applications to use only big packages.

db2cli.ini keyword syntax:

OnlyUseBigPackages= **0** | **1**

Default setting:

CLI applications use small packages (3 by default) with 64 sections, and once these sections are exhausted, it starts using big packages, with 384 sections.

Usage notes:

This option can be used to allow the applications to access to a large number of sections (384) in big packages, as opposed to 64 sections in small packages.

When the **OnlyUseBigPackages**=1 keyword is set, the small CLI packages should not be used.

OptimizeForNRows CLI/ODBC configuration keyword

Appends 'OPTIMIZE FOR n ROWS' clause to every select statement.

db2cli.ini keyword syntax:

OptimizeForNRows = *integer*

Default setting:

The clause is not appended.

Equivalent statement attribute:

SQL_ATTR_OPTIMIZE_FOR_NROWS

Usage notes:

This option will append the "OPTIMIZE FOR n ROWS" clause to every select statement, where n is an integer larger than 0. If set to 0 (the default) this clause will not be appended.

PWD CLI/ODBC configuration keyword

Defines the default password.

db2cli.ini keyword syntax:

PWD = *password*

Default setting:

None

Usage notes:

Use this *password* if a password is not provided by the application at connect time.

The password is stored as plain text in the `db2cli.ini` file and is therefore not secure.

A password phrase can be used as the password when accessing DB2 for z/OS servers. A password phrase is a character string consisting of mixed-case letters, numbers, and special characters including blanks. Password phrases have security

advantages over passwords in that they are long enough to withstand most hacking attempts yet are unlikely to be written down because they are so easy to remember.

PWDPlugin CLI/ODBC configuration keyword

Specifies the name of the userid-password plug-in library to be used for client side authentication for file DSN or DSN-less connectivity.

db2cli.ini keyword syntax:

PWDPlugin = *plug-in name*

Default setting:

By default, the value is null and the DB2 supplied userid-password plug-in library is used.

Usage notes:

This can be set in the `[Data Source]` section of the `db2cli.ini` file for the given data source, or in a connection string.

This parameter specifies the name of the userid-password plug-in library to be used for client-side connection authentication. The plug-in is used when the client is authenticated using SERVER or SERVER_ENCRYPT authentication.

Patch1 CLI/ODBC configuration keyword

Specifies a work-around for known CLI/ODBC application problems.

db2cli.ini keyword syntax:

Patch1 = { **0** | **1** | **2** | **4** | **8** | **16** | **...** }

Default setting:

Use no work-arounds.

Usage notes:

This keyword is used to specify a work-around for known problems with ODBC applications. The value specified can be for none, one, or multiple work-arounds. The patch values specified here are used in conjunction with any Patch2 values that might also be set.

Using the CLI/ODBC Settings notebook you can select one or more patches to use. If you set the values in the `db2cli.ini` file itself and want to use multiple patch values then simply add the values together to form the keyword value. For example, if you want the patches 1, 4, and 8, then specify Patch1=13.

- 0 = No work around (default)

Patch1 CLI/ODBC configuration keyword

Table 163. Patch1 CLI/ODBC configuration keyword values

Value	Description
1	Replaces `COUNT(exp)` with `COUNT(*)` before sending the SQL statement to the database server. DB2 for z/OS Version 7 and Version 8.1 in "compatibility mode" do not support the syntax `COUNT(exp)`. With PATCH1 = 1 specified, applications that use this syntax can run without modification. Depending on the complexity of the query that is being replaced, this replacement might not produce the expected results. This setting is ignored for DB2 Database for Linux, UNIX, and Windows, and for versions of DB2 for z/OS later than Version 8.1.
4	Changes input timestamp data to date data if the time and fraction part of the timestamp are zero. For example, {ts 'YYYY-MM-DD 00:00:00'} is changed to {d 'YYYY-MM-DD'}. This value is typically needed for older versions of Microsoft Access.
8	Changes input timestamp data to time data if the date part of the timestamp is either 1899-12-30 or 1900-01-01. For example, {ts '1899-12-30 HH:MM:SS'} is changed to {t 'HH:MM:SS'}. This value is typically needed for older versions of Microsoft Access.
64	Null-terminates output GRAPHIC strings. This value is typically needed by Microsoft Access in a double-byte (DBCS) environment.

Table 163. Patch1 CLI/ODBC configuration keyword values (continued)

Value	Description
128	Disables the default performance optimization behavior for the MSysConf table associated with some Microsoft applications. Microsoft applications, such as Microsoft Access, use a configuration table called MSysConf. Once these applications successfully connect to a database, they will typically issue the following query: "SELECT Config, nValue FROM MSysConf". Because the MSysConf table does not exist in a DB2 database by default, this query fails with the error "SQL0204N "MSysConf" is an undefined name.". Microsoft applications can handle this error and continue processing, however, issuing the query across the network to the DB2 server incurs various resource usage. To enhance performance, CLI assumes that this query will always fail, so when it detects that an application is trying to execute this query, it automatically returns an error with an SQLSTATE of S0002 (Table not found). The query, therefore, is never sent to the server. If, however, the user has created the MSysConf configuration table in the database and wants the application to access it, this PATCH1 value can be set to disable the performance optimization and allow the query to be executed.
256	Service use only
512	Service use only
1024	Returns SQL_SUCCESS_WITH_INFO instead of SQL_NO_DATA_FOUND from the SQLExecute() and SQLExecDirect() functions if the executed UPDATE or DELETE statement affected no rows. This value might be needed by some Microsoft Visual Basic applications.
4096	Prevents a COMMIT from being issued after closing a cursor in autocommit mode.
8192	Returns an extra result set after invoking a stored procedure. This extra result set has one row and consists of the output values of the stored procedure. This PATCH1 value might be needed by some Powerbuilder applications that require an extra result set.
32768	Forces the driver to make Microsoft Query applications work with DB2 for z/OS synonyms.
65536	Deprecated
131072	Deprecated

Table 163. Patch1 CLI/ODBC configuration keyword values (continued)

Value	Description
262144	Deprecated
524288	Deprecated
1048576	Service use only
2097152	Service use only

Patch2 CLI/ODBC configuration keyword

Specifies a workaround for known CLI and ODBC application problems.

db2cli.ini keyword syntax:

Patch2 = *"patch value 1, patch value 2, patch value 3, ..."*

Default setting:

Use no workarounds

Usage notes:

You can specify this value for zero, one, or multiple workarounds. You can use the patch values that are specified here with any **Patch1** values that might also be set.

When specifying multiple patches, the values are specified in a comma-delimited string (unlike the **Patch1** option where the values are added together and the sum is used).

- `0` = No work around (default)

To set **Patch2** values `3`, `4` and `8` you would specify:

```
Patch2="3, 4, 8"
```

Table 164. Patch2 CLI/ODBC configuration keyword values

Value	Description
1	Deprecated
3	Service use only.
4	Deprecated
5	Deprecated
6	Forces CLI to return a message indicating that scrollable cursors are not supported. This setting is required by some applications (such as Visual Basic) that make use of LOBs or that do not require or want scrollable cursors to be used, even though they have been explicitly requested by the application.
7	Maps all GRAPHIC column data types to the CHAR column data type. The precision of a GRAPHIC column will also be doubled; for example, GRAPHIC(20) will be reported as CHAR(40).
8	Ignores catalog search arguments in schema calls.
11	SQLGetInfo() function reports that catalog names are supported for Visual Basic stored procedures.

Table 164. Patch2 CLI/ODBC configuration keyword values (continued)

Value	Description
12	Deprecated
13	Prevents keywords in the `db2cli.ini` initialization file from being appended to the output connection string.
14	Deprecated
15	Causes a period separator to be used instead of the default locale's decimal separator in character output.
16	Deprecated
17	Deprecated
18	Attempts to replace literals with parameter markers for inefficient applications that use literals repeatedly. It is only applicable to INSERT SQL statements with the VALUES clause using only literals. Coding your application properly to use parameter markers is the best solution. This value is no longer available in Version 9.7 or later.
19	Removes parentheses from the ON clause of an outer join, where the outer join is an ODBC escape sequence and the server is DB2 for MVS/ESA Version 5. DB2 for MVS/ESA Version 5 does not currently support the ODBC syntax where parentheses are permitted in the ON clause of an outer join clause. Setting this **Patch2** value allows the outer join escape sequence to be used against DB2 for MVS/ESA Version 5. You should only set this value when the server is DB2 for MVS/ESA Version 5.
20	Forces CLI to rewrite the BETWEEN predicate when the server is DB2 for MVS/ESA. DB2 for MVS/ESA does not currently support the BETWEEN predicate with parameter markers as both operands. Setting this **Patch2** value causes (expression ? BETWEEN ?) to be rewritten as (expression >= ? and expression <= ?).
21	Deprecated
22	Causes theSQLGetInfo() function to report `SQL_OUTER_JOINS=NO` and `SQL_OJ_CAPABILITIES=0`. This prevents the application from using outer joins where they are not supported, thus ensuring that the outer join queries do not fail.
23	Deprecated
24	Reports TIME data as SQL_CHAR data. This patch value is used as a workaround for Microsoft Access applications.

Table 164. Patch2 CLI/ODBC configuration keyword values (continued)

Value	Description
25	Removes trailing zeros in the CHAR representation of DECIMAL columns; used as a workaround for Microsoft Access applications.
28	Deprecated
29	Removes leading zeros in the string representation of DECIMAL values x, where 1 > x > -1; used as a workaround for ADO applications with some MDAC versions.
30	Disables stored procedure caching optimization.
31	Deprecated
32	Deprecated
33	Returns the ISO version of timestamp data when converted to CHAR, rather than the ODBC version.
34	Deprecated
38	Turns statement caching off.
42	Prevents the FOR UPDATE clause from being used with keyset cursors. By default, most applications behaves as though keyset cursors is updatable. However, if updatable cursor is not required, then this **Patch2** value makes the cursor read-only (but still scrollable and sensitive to changes made by others).
50	Frees LOB locators when the SQLFetch() function is executed, rather than when a COMMIT is issued. This **Patch2** value frees the locators that are used internally when applications fetch LOB data without binding the LOB columns with the SQLBindCol() function(or equivalent descriptor APIs). Locators that are explicitly returned to the application must still be freed by the application. You can use this **Patch2** value to avoid scenarios where an application receives SQLCODE = -429 (no more locators).
56	Allows client support for Early Close Cursors for those servers that do not support it as in the case of DB2 Universal Database™ for z/OS and OS/390 version 7 or earlier.
57	Allows calling a stored procedure that returns a NULL output parameter value without providing an output indicator pointer. This is normally applicable to older versions of Borland Delphi products.

Table 164. Patch2 CLI/ODBC configuration keyword values (continued)

Value	Description
58	DateTime values inserted into the database that cause truncation errors can be downgraded to a truncation warning using this **Patch2** value.
61	When data is given to the client from an SQL_CHAR data type,there might be right padded spaces. This patch value strips off right padded single byte spaces, but not double byte spaces. This behavior partially mimics the Neon Shadow Driver behavior
66	Allows applications to retrieve the regional setting that affects decimal separators in a Windows environment. The regional setting is normally ignored by default. This patch value is ignored if **Patch2**=15 or **db2set** registry variables **DB2TERRITORY** or **DB2CODEPAGE** are set. The only supported decimal separators are the period and comma.
78	Alters the behavior of the SQLGetPosition() function when the source LOB value is in a DBCLOB column on DB2 Universal Database for z/OS and OS/390 Version 7.1 or later. Setting this **PATCH2** value causes the SQLGetPosition() function to query SYSIBM.SYSDUMMYU instead of SYSIBM.SYSDUMMY1.
81	According to the CLI specification, IBM Data Server Driver for ODBC and CLI should return the column ordinal number for an expression. By default, CLI returns the column ordinal number. According to the ODBC Specification, an ODBC driver should return an empty string as the column names for an expression. If the **Patch2** CLI/ODBC configuration keyword is set to 81, an empty string is returned for the column name of an expression by the IBM Data Server Driver for ODBC and CLI.
82	Forces the CLI to use the meaning of the SQL_ATTR_REPLACE_QUOTED_LITERALS value instead of the SQL_ATTR_RESET_CONNECTION value.

Port CLI/ODBC configuration keyword

Specifies the server system's service name or port number, used with a file DSN or in a DSN-less connection.

db2cli.ini keyword syntax:

Port = *service name* | *port number*

Default setting:

None

Only applicable when:
Protocol set to TCPIP

Usage notes:

Use this option in conjunction with the Hostname option to specify the required attributes for a TCP/IP connection from this client machine to a server running DB2. These two values are only considered when the Protocol option is set to TCPIP.

Specify either the server system's service name or its port number. The service name must be available for lookup at the client machine if it is used.

ProgramID CLI/ODBC configuration keyword

Sets the user-defined character string, with a maximum length of 80 bytes, that associates an application with a connection. Once this attribute is set, DB2 for z/OS Version 8 and later associates this identifier with any statements inserted into the dynamic SQL statement cache.

db2cli.ini keyword syntax:
ProgramID = **<string>**

Default setting:
none

Equivalent connection attribute:
SQL_ATTR_INFO_PROGRAMID

Usage notes:
When monitoring a CLI application, you can use the **ProgramID** to identify a statement which was inserted into the dynamic SQL statement cache. The **ProgamID** allows you to specify an identifier, as a string up to 80 bytes, at the connection and statement level. If the **ProgamID** is set, both `SQLGetConnectAttr()` and `SQLGetStatementAttr()` APIs returns the **ProgamID** value specified.

Note: This attribute is only supported for CLI applications accessing DB2 UDB for z/OS Version 8 and later or IBM Informix® Dynamic Servers (IDS).

ProgramName CLI/ODBC configuration keyword

Sets the default client application name to a user-defined name which is used to identify the application at the server when monitoring.

db2cli.ini keyword syntax:
ProgramName = **<string>** | **PID**

Default setting:
The application is identified by the client. By default, the first 20 bytes of the executable name is used.

Equivalent connection attribute:
SQL_ATTR_INFO_PROGRAMNAME

Usage notes:

When monitoring a CLI application, it may be useful to identify the application by a user-defined string, instead of by the default identifier that DB2 assigns. ProgramName allows the user to specify the identifier as either a string up to 20 bytes in length or the string "PID" (without the quotation marks).

If ProgramName is set to "PID" for a CLI application, the application's name will consist of the prefix "CLI" along with the application's process ID and the current active connection handle, as follows: CLI<pid>:<connectionHandle#>. The "PID" setting is useful when monitoring application servers that run multiple applications with numerous connections to the same database.

(When the ProgramName keyword is set to "PID" for other types of applications, the "CLI" prefix is replaced with the following values corresponding to the type of application: "JDBC" for JDBC applications, "OLEDB" for OLE DB applications, and "ADONET" for .NET applications.)

PromoteLONGVARtoLOB CLI/ODBC configuration keyword

Promotes LONGVARxxx data types to xLOB data types.

db2cli.ini keyword syntax:

PROMOTELONGVARTOLOB = **0** | **1**

Default setting:

0

Usage notes:

This option should only be used when a LONGVARxxxx value has the potential to exceed 32K. The return type of the column is promoted to xLOB to hold the larger data size.

Protocol CLI/ODBC configuration keyword

Communications protocol used for File DSN or in a DSN-less connection.

db2cli.ini keyword syntax:

Protocol = **TCPIP** | **TCPIP6** | **TCPIP4** | **IPC** | **LOCAL**

Default setting:

none

Usage notes:

This can be set in the [Data Source] section of the db2cli.ini file for the given data source, or in a connection string.

TCP/IP is the only protocol supported when using a File DSN. Set the option to the string TCPIP (without the slash).

When this option is set then the following options must also be set:

- Database;
- ServiceName; and
- Hostname.

IPC connectivity can be specified by setting Protocol to either **IPC** or **LOCAL**.

When Protocol = **IPC** | **LOCAL** the Instance keyword must also be set.

QueryTimeoutInterval CLI/ODBC configuration keyword

Delay (in seconds) between checks for a query timeout.

db2cli.ini Keyword Syntax:

QueryTimeoutInterval = **0** | **5** | **positive integer**

Default Setting:

5 seconds

Usage Notes:

An application can use the `SQLSetStmtAttr()` function to set the **SQL_ATTR_QUERY_TIMEOUT** statement attribute. This attribute indicates the number of seconds to wait for an SQL statement or XQuery expression to complete executing before attempting to cancel the execution and returning to the application.

The **QueryTimeoutInterval** configuration keyword is used to indicate how long the CLI driver should wait between checks to see if the query has completed.

For instance, suppose **SQL_ATTR_QUERY_TIMEOUT** is set to 25 seconds (timeout after waiting for 25 seconds), and **QueryTimeoutInterval** is set to 10 seconds (check the query every 10 seconds). The query will not time out until 30 seconds (the first check AFTER the 25 second limit). CLI implements query timeout by starting a thread that periodically queries the status of each executing query. The **QueryTimeoutInterval** value specifies how long the query timeout thread waits between checks for expired queries. Because this is an asynchronous operation to the queries being executed, it is possible that a given query may not be timed out until **SQL_ATTR_QUERY_TIMEOUT** + **QueryTimeoutInterval** seconds. In the example, the best-case timeout would be at 26 seconds, and the worst-case timeout would be at 35 seconds.

There may be cases where the **SQL_ATTR_QUERY_TIMEOUT** is set to a value which is too low, and the query should NOT be timed-out. If the application cannot be modified (that is, a third party ODBC application), then the **QueryTimeoutInterval** can be set to 0, and the CLI driver will ignore the **SQL_ATTR_QUERY_TIMEOUT** setting, and therefore wait for SQL statements to complete execution before returning to the application.

If **QueryTimeoutInterval** is set to 0, any attempt by the application to set **SQL_ATTR_QUERY_TIMEOUT** will result in SQLSTATE 01S02 (Option Value Changed).

This option is only valid in the **COMMON** section of a initialization file and therefore applies to all connections to DB2 database(s).

Alternatively, **QueryTimeoutInterval** can be set to a value that is larger than the **SQL_ATTR_QUERY_TIMEOUT** setting, thus preventing timeouts from occurring at the specified interval. For example, if the application sets a 15 second **SQL_ATTR_QUERY_TIMEOUT** value, but the server requires at least 30 seconds to execute the query, the **QueryTimeoutInterval** can be set to a value of 30 seconds or so to prevent this query from timing out after 15 seconds.

Note:

- The .NET Framework does not support settings in `db2cli.ini` file.
- This CLI keyword is ignored if it is used inside a stored procedure or routine that uses CLI API calls.

- The **QueryTimeoutInterval** can also interrupt a LOAD, which returns SQL3005N instead of SQL0952N.

ReadCommonSectionOnNullConnect CLI/ODBC configuration keyword

Allows a NULL connect to process the COMMON section of the db2cli.ini initialization file.

db2cli.ini keyword syntax:

ReadCommonSectionOnNullConnect = **0** | **1**

Default setting:

A NULL connect does not process the db2cli.ini initialization file.

Usage notes:

For use with DB2 CLI and DB2 .NET stored procedures, specify 1 to allow stored procedures to read the COMMON section of the db2cli.ini file, thus allowing stored procedures to use keywords listed in that section.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2 databases.)

ReceiveTimeout CLI/ODBC configuration keyword

Specifies the time in seconds to wait for a reply from the server on an established connection before terminating the attempt and generating a communication timeout error.

db2cli.ini keyword syntax:

ReceiveTimeout = **0** | **1** |**2** | **...** | **32767**

Default setting:

The client waits indefinitely for a reply from the server on an established connection.

Equivalent connection attribute:

SQL_ATTR_RECEIVE_TIMEOUT

Usage notes:

This keyword has no effect during connection establishment and is only supported for TCP/IP protocol.

Reopt CLI/ODBC configuration keyword

Enables query optimization or reoptimization of SQL statements that have special registers, global variables, or parameter markers.

db2cli.ini keyword syntax:

Reopt = **2** | **3** | **4**

Default setting:

No query optimization occurs at query execution time. The default estimates chosen by the compiler are used for special registers, global variables, or parameter markers.

Equivalent connection or statement attribute:

SQL_ATTR_REOPT

Usage notes:

Optimization occurs by using the values available at query execution time for the special registers, global variables, or parameter markers instead of the default estimates that are chosen by the compiler. The valid values of the keyword are:

- 2 = SQL_REOPT_NONE. This is the default. No query optimization occurs at query execution time. The default estimates chosen by the compiler are used for the special registers, global variables, or parameter markers. The default NULLID package set is used to execute dynamic SQL statements.
- 3 = SQL_REOPT_ONCE. Query optimization occurs once at query execution time, when the query is executed for the first time. The NULLIDR1 package set, which is bound with the REOPT ONCE bind option, is used.
- 4 = SQL_REOPT_ALWAYS. Query optimization or reoptimization occurs at query execution time every time the query is executed. The NULLIDRA package set, which is bound with the REOPT ALWAYS bind option, is used.

The NULLIDR1 and NULLIDRA are reserved package set names, and when used, REOPT ONCE and REOPT ALWAYS are implied respectively. These package sets have to be explicitly created with the following commands:

```
db2 bind db2clipk.bnd collection NULLIDR1
db2 bind db2clipk.bnd collection NULLIDRA
```

If both the Reopt and CurrentPackageSet keywords are specified, CurrentPackageSet takes precedence.

ReportPublicPrivileges CLI/ODBC configuration keyword

Reports PUBLIC privileges in `SQLColumnPrivileges()` and `SQLTablePrivileges()` results.

db2cli.ini keyword syntax:

ReportPublicPrivileges = **0** | **1**

Default setting:

PUBLIC privileges are not reported.

Usage notes:

This keyword specifies if privileges assigned to the PUBLIC group are to be reported as if PUBLIC was a user in the `SQLColumnPrivileges()` and `SQLTablePrivileges()` results.

ReportRetryErrorsAsWarnings CLI/ODBC configuration keyword

Returns errors that were uncovered during CLI error recovery as warnings.

db2cli.ini keyword syntax:

ReportRetryErrorsAsWarnings = **0** | **1**

Only applicable when:

RetryOnError keyword is set to 1.

Default setting:

Do not return errors uncovered during CLI error recovery to the application.

Usage notes:

By default, when the CLI retry logic is able to recover successfully from a non-fatal error, it masks that error from the application by returning SQL_SUCCESS. Because application binding errors can be hidden this way, for debugging purposes, you may want to set ReportRetryErrorsAsWarnings to 1. This setting keeps the error recovery on, but forces CLI to return to the application, any errors that were uncovered as warnings.

RetCatalogAsCurrServer CLI/ODBC configuration keyword

Specifies whether catalog functions return the CURRENT SERVER value or a null value for the catalog columns.

db2cli.ini keyword syntax:

RetCatalogAsCurrServer= **0** | 1

Default setting:

If the target DBMS returns null for the catalog columns, the CURRENT SERVER value will not be substituted.

Usage notes:

If the catalog functions for the target DBMS return a null value for the catalog columns, setting RetCatalogAsCurrServer to 1 causes the DBMS to return the CURRENT SERVER value instead.

- 0 = Catalog functions return the null value for the catalog columns (default).
- 1 = Catalog functions return the CURRENT SERVER value, instead of the null value, for the catalog columns.

For example, assume the catalog function `SQLTables()` returns a result set where the values in the TABLE_CAT column are null values. Setting RetCatalogAsCurrServer to 1 causes the DBMS to return the CURRENT SERVER value in the TABLE_CAT column.

RetOleDbConnStr CLI/ODBC configuration keyword

Specifies whether the Mode CLI/ODBC configuration keyword returns a numeric value or string value.

db2cli.ini keyword syntax:

RetOleDbConnStr = **0** | **1**

Default setting:

The value for the Mode CLI/ODBC configuration keyword is returned as a string.

Usage notes:

The Mode CLI/ODBC configuration keyword sets the CONNECT mode to either SHARE or EXCLUSIVE. OLE DB expects the value for Mode to have a numeric representation instead of a string representation. RetOleDbConnStr toggles between returning a string and a numeric value.

The possible settings are as follows:

- 0 — the value returned by `SQLDriverConnect()` and `SQLBrowseConnect()`for the Mode keyword is either SHARE or EXCLUSIVE

- 1 — the value returned by SQLDriverConnect() and SQLBrowseConnect()for the Mode keyword is either 3 (for SHARE) or 12 (for EXCLUSIVE)

For example, if you set RetOleDbConnStr=1 and call SQLDriverConnect()or SQLBrowseConnect() with the following input connection string for a shared connection:

```
DSN=SAMPLE;MODE=SHARE
```

then the output connection string will have the following format:

```
DSN=SAMPLE;UID=;PWD=;MODE=3
```

If you set RetOleDbConnStr=1 and call SQLDriverConnect()or SQLBrowseConnect() with the following input connection string for an exclusive connection:

```
DSN=SAMPLE;UID=NEWTON;PWD=SECRET;MODE=EXCLUSIVE
```

then the output connection string will have the following format:

```
DSN=SAMPLE;UID=NEWTON;PWD=SECRET;MODE=12
```

OLE DB applications that use the string representation for the value of the Mode keyword returned by SQLDriverConnect()and SQLBrowseConnect() will receive an error from OLE DB Component Services. OLE DB Component Services returns an error because it expects the keyword Mode to have numeric values. Setting RetOleDbConnStr to 1 avoids this behavior, as the value for Mode will then be numeric.

RetryOnError CLI/ODBC configuration keyword

Turns on or off the CLI driver's error recovery behavior.

db2cli.ini keyword syntax:

RetryOnError = **0** | **1**

Default setting:

Allow the CLI driver to attempt error recovery on non-fatal errors.

Usage notes:

By default, CLI will attempt to recover from non-fatal errors, such as incorrect binding of application parameters, by retrieving additional information about the failing SQL statement and then executing the statement again. The additional information retrieved includes input parameter information from the database catalog tables. If CLI is able to recover successfully from the error, by default, it does not report the error to the application. The CLI/ODBC configuration keyword ReportRetryErrorsAsWarnings allows you to set whether error recovery warnings are returned to the application or not.

Important: Once CLI has successfully completed the error recovery, the application may behave differently, because CLI will use the catalog information gathered during the recovery for subsequent executions of that particular SQL statement, rather than the information provided in the original SQLBindParameter() function calls. If you do not want this behavior, set RetryOnError to 0, forcing CLI not to attempt recovery. You should, however, modify the application to correctly bind statement parameters.

ReturnAliases CLI/ODBC configuration keyword

Controls whether the CLI schema APIs report aliases in the metadata results.

db2cli.ini keyword syntax:

ReturnAliases = **0** | **1**

Default setting:

1: By default, aliases will be considered when qualifying rows for metadata procedures.

Usage notes:

This keyword specifies whether aliases (or synonyms) should be considered when qualifying rows for metadata procedures. Not considering aliases can provide significant performance benefits by avoided costly joins with the base tables to determine the addition tables that should qualify for a given query.

- `0` : Aliases will not be considered when qualifying rows for metadata procedures (better performance.)
- `1` : Aliases will be considered when qualifying rows for metadata procedures.

The following CLI APIs are affected by this keyword :

- `SQLColumns()`
- `SQLColumnPrivileges()`
- `SQLTables()`
- `SQLTablePrivileges()`
- `SQLStatistics()`
- `SQLSpecialColumns()`
- `SQLForeignKeys()`
- `SQLPrimaryKeys()`

ReturnSynonymSchema CLI/ODBC configuration keyword

Controls whether CLI schema APIs report the schema name for DB2 for z/OS synonyms in the TABLE_SCHEM column of the schema procedure result sets.

db2cli.ini keyword syntax:

ReturnSynonymSchema = **0** | **1**

Default setting:

1: By default, the creator of the synonym will be returned in the TABLE_SCHEM column.

Usage notes:

Valid settings:

- `0` : the TABLE_SCHEM column of the procedure result set will be NULL.
- `1` : the TABLE_SCHEM column of the procedure result set will contain the creator of the synonym.

You cannot access a synonym on a DB2 for z/OS server using a name qualified with a schema. For this reason, the meaning of the TABLE_SCHEM column of a CLI schema API result set is different, with respect to synonyms, when you are running against a DB2 for z/OS server.

This CLI keyword has no effect when you use CLI schema APIs against a DB2 Database for Linux, UNIX, and Windows server.

The following CLI APIs are affected by this keyword:

- SQLColumns()
- SQLColumnPrivileges()
- SQLTables()
- SQLTablePrivileges()
- SQLStatistics()
- SQLSpecialColumns()
- SQLForeignKeys()
- SQLPrimaryKeys()

You must have the following program temporary fixes (PTFs) on the DB2 for z/OS database server to use this keyword:

Table 165. DB2 for z/OS PTFs for **ReturnSynonymSchema**

DB2 for z/OS	PTF or APAR numbers
Version 7	UK13643
Version 8	UK13644
Version 9	

SQLOverrideFileName CLI/ODBC configuration keyword

Specifies the location of the override file, which lists CLI statement attribute settings for particular SQL statements.

db2cli.ini keyword syntax:

SQLOverrideFileName = **<absolute or relative path name>**

Default setting:

No override file is used.

Usage notes:

The **SQLOverrideFileName** keyword specifies the location of the override file to be read by the CLI driver. An override file contains values for CLI statement attributes that apply to particular SQL statements. For example, you can specify the **SQLOverrideFileName** keyword with path and the override file name in the db2cli.ini file.

```
[MyDatabase]
SQLOverrideFileName=C:\temp\myfile.txt
```

In the c:\temp\ path, you would have override file named myfile.txt, containing values for CLI statement attributes that applies to particular SQL statements. The override file would start with a COMMON section ([COMMON]) containing only one keyword called **Stmts**, which tells you how many statements to override. Following sample override file (myfile.txt) has two statements:

```
[Common]
Stmts=2

[1]
StmtIn=SELECT * FROM Employee
StmtAttr=SQL_ATTR_BLOCK_FOR_NROWS=50;SQL_ATTR_OPTIMIZE_FOR_NROWS=1;
```

```
[2]
StmtIn=SELECT * FROM Sales
StmtAttr=SQL_ATTR_MAX_ROWS=25;
```

The number specified by **Stmts** in the [COMMON] section of the override file equals the number of SQL statements contained in the override file.

SaveFile CLI/ODBC configuration keyword

Specifies the file name of a DSN file in which to save the attribute values of the keywords used in making the present, successful connection.

db2cli.ini keyword syntax:

You can not set this keyword in the db2cli.ini file.

You can specify the value of this keyword in the connection string in SQLDriverConnect like this:

```
SaveFile = file name
```

SchemaList CLI/ODBC configuration keyword

Restricts schemas that are used to query table information.

db2cli.ini keyword syntax:

SchemaList = " *'schema1'*, *'schema2'*,... *'schemaN'* "

Default setting:

None

Usage notes:

Use the **SchemaList** keyword to provide a more restrictive default, and therefore improve performance, for applications that list every table in the DBMS.

If there are a large number of tables that are defined in the database, you can specify a schema list to reduce the time that it takes for the application to query table information, and reduce the number of tables that are listed by the application. Each schema name is case-sensitive, must be delimited with single quotation marks, and separated by commas. You must also enclose the entire string in double quotation marks. For example:

```
SchemaList="'USER1','USER2','USER3'"
```

For DB2 for z/OS, you can also include CURRENT SQLID in this list, but without the single quotation marks, for example:

```
SchemaList="'USER1',CURRENT SQLID,'USER3'"
```

The maximum length of the string is 256 characters.

You can use this keyword with **DBName** and **TableType** to further limit the number of tables for which information is returned.

You can specify **USRLIBL* or **ALL* along with the existing list of schema names to resolve unqualified stored procedure calls or to find libraries in catalog API calls. *ALL, CLI searches on all existing schemas in the connected database. *ALL is the default for CLI. For DB2 for i servers, if you specify *USRLIBL, CLI searches on the current libraries of the server job.

If you are migrating from the IBM i Access ODBC driver and you specified *USRLIBL in the DBQ or DefaultLibraries connection string keywords, add *USRLIBL to the list of schema names in the **SchemaList** keyword as shown in the example:

```
[DSNSAMP]
SCHEMALIST="*USRLIBL"
```

security CLI/ODBC configuration keyword

Specifies whether or not SSL support is used for File DSN or in a DSN-less connection.

db2cli.ini keyword syntax:

security = SSL

Default setting:

None.

Usage notes:

This can be set in the [Data Source] section of the db2cli.ini file for the given data source, or in a connection string.

This parameter specifies whether a TCP/IP communication will be with SSL support or not. It can only be used with the protocols TCPIP, TCPIP4, or TCPIP6. If not specified, a normal TCP/IP without SSL support will be used.

ServerMsgMask CLI/ODBC configuration keyword

Specifies when CLI should request the error message from the server.

db2cli.ini keyword syntax:

ServerMsgMask = **0** | **1** | **-2** | **-1**

Default setting:

CLI will check the local message files first to see if the message can be retrieved. If no matching SQLCODE is found, CLI will request the information from the server.

Equivalent connection attribute:

SQL_ATTR_SERVER_MSGTXT_MASK

Usage notes:

This keyword is used in conjunction with the "UseServerMsgSP CLI/ODBC configuration keyword" on page 424. The keyword can be set to:

- **0 (default)** = SQL_ATTR_SERVER_MSGTXT_MASK_LOCAL_FIRST. CLI will check the local message files first to see if the message can be retrieved. If no matching SQLCODE is found, CLI will request the information from the server.
- 1 = SQL_ATTR_SERVER_MSGTXT_MASK_WARNINGS. CLI always requests the message information from the server for warnings but error messages are retrieved from the local message files.
- -2 = SQL_ATTR_SERVER_MSGTXT_MASK_ERRORS. CLI always requests the message information from the server for errors but warning messages are retrieved from the local message files.

- -1 = SQL_ATTR_SERVER_MSGTXT_MASK_ALL. CLI always requests the message information from the server for both error and warning messages.

ServiceName CLI/ODBC configuration keyword

The server system's service name or port number, used with file DSN or in a DSN-less connection.

db2cli.ini keyword syntax:
ServiceName = *service name* | *port number*

Default setting:
None

Only applicable when:
Protocol set to TCPIP

Usage notes:

Use this option in conjunction with the Hostname option to specify the required attributes for a TCP/IP connection from this client machine to a server running DB2. These two values are only considered when the PROTOCOL option is set to TCPIP.

Specify either the server system's service name or its port number. The service name must be available for lookup at the client machine if it is used.

SkipTrace CLI/ODBC configuration keyword

Excludes CLI trace information from the DB2 trace.

db2cli.ini keyword syntax:
SkipTrace = **0** | **1**

Default setting:
Do not skip the trace function.

Usage notes:

This keyword can improve performance by allowing the DB2 trace function to bypass CLI applications. Therefore, if the DB2 trace facility db2trc is turned on and this keyword is set to 1, the trace will not contain information from the execution of the CLI application.

Turning SkipTrace on is recommended for production environments on the UNIX platform where trace information is not required. Test environments may benefit, however, from having trace output, so this keyword can be turned off (or left at its Default setting) when detailed execution information is desired.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

SQLCODEMAP CLI/ODBC configuration keyword

Specifies whether SQLCODE mapping is used or turned off.

db2cli.ini keyword syntax:
SQLCODEMAP = <*MAP*> | <*NOMAP*>

Default setting:
MAP

Usage notes:
You can set this keyword in the [Data Source] section of the db2cli.ini file, or in the connection string.

If the value of the keyword is MAP, SQLCODE mapping is used. If you specify the NOMAP option, SQLCODE mapping is turned off.

SSLClientLabel CLI/ODBC configuration keyword

Specifies a unique SSL label that is mapped to a specific certificate to use with CERTIFICATE authentication.

db2cli.ini keyword syntax:
SSLClientLabel = <*label*>

Default setting:
None.

Usage notes:
The **SSLClientLabel** keyword in only available for use with CERTIFICATE authentication starting in DB2 Version 9.7 Fix Pack 6 and later. You can set this keyword can be set in the [Data Source] section of the db2cli.ini file, or in the connection string.

When you configure certificate-based authentication to supply authentication information, you must specify the **SSLClientLabel** keyword in the db2dsdriver.cfg configuration file or in the db2cli.ini configuration file.

If you set the authentication parameter to the CERTIFICATE option but do not specify the **SSLClientLabel** keyword in the db2cli.ini configuration file, the db2dsdriver.cfg configuration file, or the connection string, error CLI0221E is returned. If you do not set the authentication parameter to the **CERTIFICATE** option but specify the **SSLClientLabel** keyword in the db2cli.ini configuration file, the db2dsdriver.cfg configuration file, or the connection string, error CLI0222E is returned.

SSLClientKeystash CLI/ODBC configuration keyword

Specifies the SSL stash file used for File DSN or in a DSN-less connection.

db2cli.ini keyword syntax:
SSLClientKeystash = <*fully qualified stash file path*>

Default setting:
None.

Usage notes:
This can be set in the [Data Source] section of the db2cli.ini file for the given data source, or in a connection string.

This parameter specifies the fully qualified path of the stash file (.sth), which stores an encrypted password to the key database file. The stash file is used to access the key database file during the SSL handshake. This parameter must be defined if the SSL protocol (**security**=SSL) is specified.

The **SSLCLientKeystash** keyword is mutually exclusive with the **SSLClientKeystoreDBPassword** keyword. If the SSL protocol (security=SSL) is specified, either **SSLClientKeystash** or **SSLClientKeystoreDBPassword** must be specified in the connection string, CLI configuration file, db2cli.ini, or in your data server driver configuration file, db2dsdriver.cfg. Otherwise, the connection fail error will be returned.

Note:

- The **ssl_client_keystash** keyword is also supported to provide compatibility with earlier version
- **SSLClientKeystash** keyword is supported starting from DB2 Version 9.7 Fix Pack 6

SSLClientKeystoredb CLI/ODBC configuration keyword

Specifies the SSL key database file that is used for File DSN or in a DSN-less connection.

db2cli.ini keyword syntax:

SSLClientKeystoredb = *<fully qualified key file path>*

Default setting:

None.

Usage notes:

This can be set in the `[Data Source]` section of the `db2cli.ini` file for the given data source, or in a connection string.

This parameter specifies the fully qualified path of the key database file (.kdb). The key database file stores the signer certificate from the server personal certificate.

- For a self-signed server personal certificate, the signer certificate is the public key of the personal certificate.
- For a certificate authority (CA) signed server personal certificate, the signer certificate is the root CA certificate of the CA that signed the personal certificate.

If the SSL protocol (**security**=SSL) is used, this parameter must be defined. The signer certificate from the server's personal certificate must also exist on the client for authentication to take place.

Note:

- The **ssl_client_keystoredb** keyword is also supported to provide compatibility with earlier version
- **SSLClientKeystoredb** keyword is supported starting from DB2 Version 9.7 Fix Pack 6

SSLClientKeystoreDBPassword CLI/ODBC configuration keyword

Specifies the password for the SSL connection when the authentication parameter is set to CERTIFICATE.

db2cli.ini keyword syntax:

SSLClientKeystoreDBPassword = *<password>*

Default setting:

None.

Usage notes:

The **SSLClientKeystoreDBPassword** keyword in only available for use with CERTIFICATE authentication starting in DB2 Version 9.7 Fix Pack 6 and later. You can set this keyword can be set in the [Data Source] section of the db2cli.ini file, or in the connection string.

The **SSLClientKeystash** and **SSLClientKeyStoreDBPassword** configuration parameters are mutually exclusive. If you specify both the **SSLClientKeystash** configuration parameter and the **SSLClientKeyStoreDBPassword** configuration parameter in either the db2cli.ini configuration file or the db2dsdriver.cfg configuration file, error CLI0219E is returned.

StaticCapFile CLI/ODBC configuration keyword

Specifies the Capture File name and optionally the path where it will be saved.

db2cli.ini keyword syntax:

StaticCapFile = < **Full file name** >

Default setting:

None - you must specify a capture file.

Only applicable when:

StaticMode is set to Capture or Match

Usage notes:

This keyword is used to specify the Capture File name and optionally the directory where it will be saved.

StaticLogFile CLI/ODBC configuration keyword

Specifies the Static Profiling Log File name and optionally the directory where it will be saved.

db2cli.ini keyword syntax:

StaticLogFile = < **Full file name** >

Default setting:

No Static Profiling Log is created. If a filename is specified without a pathname then the current path will be used.

Only applicable when:

StaticMode is set to Capture or Match

Usage notes:

This keyword is used to specify the Static Profiling Log File name and optionally the directory where it will be saved.

StaticMode CLI/ODBC configuration keyword

Specifies whether the CLI/ODBC application will capture SQL or use a static SQL Package for this DSN.

db2cli.ini keyword syntax:

StaticMode = **DISABLED** | **CAPTURE** | **MATCH**

Default setting:
Disabled - SQL statements are not captured and no static SQL package is used.

Usage notes:

This option allows you to specify how the SQL issued by the CLI/ODBC application for this DSN will be processed:

- DISABLED = Static mode disabled. No special processing. The CLI/ODBC statements will be executed as dynamic SQL with no change. This is the default.
- CAPTURE = Capture Mode. Execute the CLI/ODBC statements as dynamic SQL. If the SQL statements are successful, they will be captured into a file (known as the Capture File) to be bound by the DB2CAP command later.
- MATCH = Match mode. Execute the CLI/ODBC statements as static SQL statements if a matching statement is found in the Capture Files specified in StaticPackage. The Capture File must first be bound by the DB2CAP command.

StaticPackage CLI/ODBC configuration keyword

Specifies the package to be used with the static profiling feature.

db2cli.ini keyword syntax:
StaticPackage = *collection_id.package_name*

Default setting:
None - you must specify a package name.

Only applicable when:
STATICMODE is set to CAPTURE

Usage notes:

This keyword is used to specify the package to be used when the application runs in Match Mode. You first need to use Capture Mode to create the Capture File.

Only the first 7 characters of the indicated package name will be used. A one-byte suffix will be added to represent each isolation level, as follows:

- 0 for Uncommitted Read (UR)
- 1 for Cursor Stability (CS)
- 2 for Read Stability (RS)
- 3 for Repeatable Read (RR)
- 4 for No Commit (NC)

StmtConcentrator CLI/ODBC configuration keyword

Starting in Version 9.7 Fix Pack 3 and later fix packs, DB2 supports this keyword, which specifies whether dynamic statements that contain literal values use the statement cache.

db2cli.ini keyword syntax:
StmtConcentrator = **OFF** | **WITHLITERALS**

Default setting:
The default behavior specified for statement concentration on the server.

Equivalent environment or connection attribute:
SQL_ATTR_STMT_CONCENTRATOR

Usage notes:
This option specifies whether dynamic statements that contain literal values use the statement cache.

- OFF - The statement concentrator is disabled.
- WITHLITERALS - The statement concentrator with literal behavior is enabled for situations that are supported by the server. For example, the statement concentrator is not enabled if the statement has parameter markers, named parameter markers, or a mix of literals, parameter markers, and named parameter markers.

When you use this attribute against DB2 for z/OS servers older than Version 10, the request is ignored.

StreamGetData CLI/ODBC configuration keyword

Optimizes data output stream for `SQLGetData()` function.

db2cli.ini keyword syntax:
StreamGetData = **0** | **1**

Default setting:
CLI buffers all the data on the client.

Equivalent connection or statement attribute:
SQL_ATTR_STREAM_GETDATA

Usage notes:
The StreamGetData keyword is ignored if Dynamic Data Format, also known as progressive streaming, is not supported by the server. For applications that do not need to buffer data and are querying data on a server that supports Dynamic Data Format, specify 1 to indicate that data buffering is not required. The CLI client will optimize the data output stream.

If StreamGetData is set to 1 and CLI cannot determine the number of bytes still available to return in the output buffer, `SQLGetData()` returns SQL_NO_TOTAL (-4) as the length when truncation occurs. Otherwise, `SQLGetData()` returns the number of bytes still available.

StreamPutData CLI/ODBC configuration keyword

Improves performance for data passed through `SQLPutData()` function calls on one statement handle, by writing data directly to the internal connection-level communication buffer.

db2cli.ini keyword syntax:
StreamPutData = **0** | **1**

Default setting:
Do not write data directly to the connection-level buffer; write to the default statement-level buffer instead.

Usage notes:

By default, CLI writes data passed in through `SQLPutData()` function calls to an internal statement-level buffer. On the subsequent `SQLParamData()` call, the contents

of the buffer are then written to an internal connection-level communication buffer and sent to the server. If only one statement handle is used to insert data into a target database on a particular connection at a given point in time, then you can improve performance by setting StreamPutData=1. This causes CLI to write the put data directly to the connection-level buffer. If, however, multiple statements concurrently insert data into a target database on a particular connection, then setting StreamPutData=1 may decrease performance and result in unexpected application errors, as the statements in the shared connection-level communication buffer will be prone to serialization.

SysSchema CLI/ODBC Configuration Keyword

Sets an alternative schema to be searched in place of the SYSIBM schema.

db2cli.ini keyword syntax:

SysSchema = **alternative schema**

Default setting:

The default table qualifier name used when querying DB2 for z/OS is SYSIBM.

Usage notes:

This option indicates an alternative schema, or table qualifier, to be searched in place of the SYSIBM schema when the CLI and ODBC Catalog Function calls are issued to obtain system catalog information from DB2 for z/OS.

Using this new schema name, the system administrator can define a set of views, or a copies of the tables, consisting of a subset of the rows for system catalog tables such as:

- SYSIBM.SYSCOLAUTH
- SYSIBM.SYSCOLUMNS
- SYSIBM.SYSDATATYPES
- SYSIBM.SYSFOREIGNKEYS
- SYSIBM.SYSINDEXES
- SYSIBM.SYSKEYS
- SYSIBM.SYSKEYCOLUSES
- SYSIBM.SYSPARMS
- SYSIBM.SYSRELS
- SYSIBM.SYSROUTINES
- SYSIBM.SYSTABAUTH
- SYSIBM.SYSTABCONST
- SYSIBM.SYSTABLES
- SYSIBM.SYSSYNONYMS

For example, if the set of views, or a copies tables, for the system catalog tables is in the ACME schema, then the view (or copy of the table) for SYSIBM.SYSTABLES is ACME.SYSTABLES; and **SysSchema** should be set to ACME.

For applications that automatically query the system catalogs for all table names, defining and using limited views of the system catalog tables reduces the number of tables listed by the application. This can reduce the time it takes for the application to query table information since a subset of table names is returned.

Defining and using copies of the system catalog tables, with the same indexes defined on the copy as those defined on the system table, can reduce the time it takes for applications to query the database.

The **SchemaList**, **TableType** and **DBName** keywords can be used in conjunction with the **SysSchema** keyword to further limit the number of tables for which information is returned.

For more information about which system catalog tables can be used with **SysSchema**, and about the function of **SysSchema**, refer to the documentation for APAR PK05102 by visiting:

Support for IBM mainframes

and searching for "PK05102".

TableType CLI/ODBC configuration keyword

Defines a default list of TABLETYPES returned when querying table information.

db2cli.ini keyword syntax:

TableType = " *'TABLE'* | *,'ALIAS'* | *,'VIEW'* | *, 'INOPERATIVE VIEW'* | *, 'SYSTEM TABLE'* | *,'SYNONYM'* "

Default setting:

No default list of TABLETYPES is defined.

Usage notes:

If there is a large number of tables defined in the database, a tabletype string can be specified to reduce the time it takes for the application to query table information, and reduce the number of tables listed by the application.

Any number of the values can be specified. Each type must be delimited with single quotation mark, separated by commas, and in uppercase. The entire string must also be enclosed in double quotation marks. For example:

```
TableType="'TABLE','VIEW'"
```

This option can be used in conjunction with DBNAME and SCHEMALIST to further limit the number of tables for which information will be returned.

TableType is used to provide a default for the CLI function that retrieves the list of tables, views, aliases, and synonyms in the database. If the application does not specify a table type on the function call, and this keyword is not used, information about all table types is returned. If the application does supply a value for the *tabletype* on the function call, then that argument value will override this keyword value.

If TableType includes any value other than TABLE, then the DBName keyword setting cannot be used to restrict information to a particular DB2 for z/OS database.

TargetPrincipal CLI/ODBC configuration keyword

Specifies the fully qualified Kerberos principal name of the DB2 instance owner for a target server.

db2cli.ini keyword syntax:

TargetPrincipal = **name/instance@REALM**

Default setting:
None

Usage notes:

For Windows 2000, Windows XP, and Windows Server 2003, the fully qualified Kerberos principal name is the DB2 server service logon account in one of the following forms: `userid@DOMAIN`, `userid@xxx.xxx.xxx.com`, or `domain\userid`. For example, if the DB2 server service account is *LocalSystem* then the TargetPrincipal is `HOST/`*`host_name`*`@DOMAIN`, where *host_name* is the fully qualified host name and *DOMAIN* is the fully qualified domain name in uppercase. Otherwise, the TargetPrincipal is `userid@DOMAIN`, where *userid* is the user ID for the DB2 server service account and *DOMAIN* is the fully qualified domain name in uppercase.

You can add this keyword to the db2cli.ini file or the db2dsdriver.cfg file. This keyword can set a data source in the `[DATA SOURCE]` or `[COMMON]` section of the db2cli.ini file.

TempDir CLI/ODBC configuration keyword

Defines the directory used for temporary files.

db2cli.ini keyword syntax:
TempDir = < *full path name* >

Default setting:
Use the system temporary directory specified by the TEMP or TMP environment variables.

Usage notes:

When working with Large Objects (CLOBS, BLOBS, and other large objects), when data conversion occurs, or when data is sent to the server in pieces, a temporary file is often created on the client machine to store the information. Using this option you can specify a location for these temporary files. The system temporary directory will be used if nothing is specified.

The keyword is placed in the data source specific section of the db2cli.ini file, and has the following syntax:

- TempDir= F:\DB2TEMP

The path specified must already exist and the user executing the application must have the appropriate authorities to write files to it. When the CLI Driver attempts to create temporary files, an SQLSTATE of HY507 will be returned if the path name is invalid, or if the temporary files cannot be created in the directory specified.

TimestampTruncErrToWarning CLI/ODBC configuration keyword

Sets the return value for an overflow of fractional seconds in TIMESTAMP.

db2cli.ini keyword syntax:
TimestampTruncErrToWarning = **0** | **1**

Default setting:
The overflow of fractional seconds in TIMESTAMP results in an error (SQLSTATE 22007).

Usage notes:

TimestampTruncErrToWarning controls whether the overflow of fractional seconds in TIMESTAMP results in an error (SQLSTATE 22007) or a warning (SQLSTATE 01S07).

Set TimestampTruncErrToWarning as follows:

- 0 - to return the default error (SQLSTATE 22007)
- 1 - to return the warning (SQLSTATE 01S07)

Trace CLI/ODBC configuration keyword

Turns on the CLI/ODBC trace facility.

db2cli.ini keyword syntax:

Trace = **0** | **1** | **db2trc**

Default setting:

No trace information is captured.

Equivalent environment attribute:

SQL_ATTR_TRACE

Usage notes:

When this option is set to value (1), CLI/ODBC trace records are appended to the file indicated by the TraceFileName configuration parameter or to files in the subdirectory indicated by the TracePathName configuration parameter. Trace CLI/ODBC configuration keyword will have no effect if neither TraceFileName or TracePathname is set.

The following example shows how to setup a CLI/ODBC trace file that is written directly to disk:

```
[COMMON]
Trace=1
TraceFileName=E:\TRACES\CLI\MONDAY.CLI
TraceFlush=1
```

When this option is set to value (db2trc), the trace facility of the DB2 instance or the DB2 Administration Server (DAS) will be started. The trace facility is controlled by the **db2trc** command and records information about operations and formats this information into readable form. Enabling the trace facility (OFF by default) might impact your system's performance. As a result, only use the trace facility when directed by a DB2 technical support representative; otherwise, turn off the trace once enough information has been recorded.

Setting the Trace CLI/ODBC configuration keyword to (db2trc) will automatically run **db2trc on** command with -cli option. To stop tracing, command `db2trc off` must be run by the user.

To create a CLI/ODBC trace file from the trace records recorded by the trace facility, the trace records must be dumped and formatted from the internal trace buffer. For example, after finishing tracing an application, the following steps should be followed to dump and format the CLI/ODBC trace:

1. `db2trc dump <dump_filename>`
2. `db2trc off`
3. `db2trc fmt -cli <dump_filename> <ODBC-CLI_trace_filename>`

Setting the Trace CLI/ODBC configuration keyword to value (db2trc) provides better performance than value (1). Also, the dumped CLI Trace file is smaller than the CLI Trace file generated with Trace value (1). Using the Trace CLI/ODBC configuration keyword value (db2trc) starts the **db2trc** trace command with a default in memory trace buffer size. With larger CLI applications, the default trace buffer size may be insufficient to prevent trace records from wrapping. To avoid this limitation, run the command **db2trc on** with the **-f** parameter to trace to a file. Here is an example command for tracing CLI to a file named `clitrc.dmp`:

```
db2trc on -f clitrc.dmp -m *.*.CLITRC.*.*
```

After tracing to a file, the trace records do not need to be dumped, but do still require formatting to obtain a final CLI Trace file.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2 databases.)

TraceAPIList CLI/ODBC configuration keyword

Specifies what APIs to trace using the CLI trace facility. If you do not set the **TraceAPIList** keyword, all APIs are traced.

To set the **TraceAPIList** keyword in the`db2cli.ini` file, issue the following command:

```
db2 update cli cfg for section common using TRACEAPILIST API ID,API ID...
```

where *API ID* is an integer that corresponds to the name of a CLI API. You can find the mapping between API names and IDs in the `/sqllib/include/sqlcli1.h` file, as shown in the following example:

```
#define  SQL_API_SQLALLOCHANDLE        1001
#define  SQL_API_SQLFREEHANDLE         1006
#define  SQL_API_SQLCLOSECURSOR        1003
#define  SQL_API_SQLENDTRAN            1005
#define  SQL_API_SQLCOLATTRIBUTE       6
#define  SQL_API_SQLGETSTMTATTR        1014
#define  SQL_API_SQLGETCONNECTATTR     1007
```

The `/sqllib/samples/cli/db2conn.c` sample program depicts the behavior of different CLI connect APIs: SQLConnect, SQLDriverConnect, and SQLBrowseConnect. If you do not set the **TraceAPIList** keyword, the CLI trace that is generated is similar to the following example:

```
[ Process: 4453, Thread: 47717036514016 ]
[ Date & Time:          06/26/2009 03:14:51.158736 ]
[ Product:              QDB2/LINUXX8664 DB2 v9.7.0.1 ]
[ Level Identifier:     08020107 ]
[ CLI Driver Version:   09.02.0000 ]
[ Informational Tokens: "DB2 v9.7.0.1","n090609","LINUXAMD6497","Fixpack 1" ]
[ Install Path:
/view/mayprasa_db2_v97fp1_linuxamd64_n090609_trc42/vbs/INST ]
[ db2cli.ini Location:  /home/mayprasa/sqllib/cfg/db2cli.ini ]
[ CLI Driver Type:      IBM DB2 Application Runtime Client ]



SQLAllocHandle( fHandleType=SQL_HANDLE_ENV, hInput=0:0,
phOutput=&00007fffb229a990 )
    ---> Time elapsed - 0 seconds

SQLAllocHandle( phOutput=0:1 )
    <--- SQL_SUCCESS   Time elapsed - +2.830000E-004 seconds
```

```
SQLSetEnvAttr( hEnv=0:1, fAttribute=SQL_ATTR_ODBC_VERSION, vParam=3,
cbParam=0 )
    ---> Time elapsed - +4.300000E-005 seconds

SQLSetEnvAttr( )
    <--- SQL_SUCCESS   Time elapsed - +1.800000E-005 seconds

SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=0:1,
phOutput=&00007fffb229a8e4 )
    ---> Time elapsed - +3.600000E-005 seconds

SQLAllocHandle( phOutput=0:1 )
    <--- SQL_SUCCESS   Time elapsed - +4.480000E-004 seconds

SQLConnect( hDbc=0:1, szDSN="sample", cbDSN=-3, szUID="", cbUID=-3, szAuthStr="",
   cbAuthStr=-3 )
    ---> Time elapsed - +2.000000E-005 seconds
( DBMS NAME="DB2/LINUXX8664", Version="09.07.0001", Fixpack="0x28020107" )
( Application Codepage=819, Database  Codepage=1208,
Database XML Codepage=1208,
   Char Send/Recv Codepage=819, Graphic Send/Recv Codepage=1200,
XML Send/Recv Codepage=1208 )

SQLConnect( )
    <--- SQL_SUCCESS   Time elapsed - +1.242704E+000 seconds
( DSN=""SAMPLE"" )
( UID=" " )
( PWD="" )
```

To trace only the SQLAllocHandle API, issue the following command:

```
db2 update cli cfg for section common using TRACEAPILIST 1001
```

where 1001 is the ID for the SQLAllocHandle API. The CLI trace that is generated is similar to the following example:

```
[ Process: 5977, Thread: 47628919556832 ]
[ Date & Time:          06/26/2009 03:17:25.066017 ]
[ Product:              QDB2/LINUXX8664 DB2 v9.7.0.1 ]
[ Level Identifier:     08020107 ]
[ CLI Driver Version:   09.02.0000 ]
[ Informational Tokens: "DB2 v9.7.0.1","n090609","LINUXAMD6497","Fixpack 1" ]
[ Install Path:
/view/mayprasa_db2_v97fp1_linuxamd64_n090609_trc42/vbs/INST ]
[ db2cli.ini Location:  /home/mayprasa/sqllib/cfg/db2cli.ini ]
[ CLI Driver Type:      IBM DB2 Application Runtime Client ]



SQLAllocHandle( fHandleType=SQL_HANDLE_ENV, hInput=0:0,
phOutput=&00007fff3657bc70 )
    ---> Time elapsed - 0 seconds

SQLAllocHandle( phOutput=0:1 )
    <--- SQL_SUCCESS   Time elapsed - +2.720000E-004 seconds

SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=0:1,
phOutput=&00007fff3657bbc4 )
    ---> Time elapsed - +8.000000E-005 seconds

SQLAllocHandle( phOutput=0:1 )
    <--- SQL_SUCCESS   Time elapsed - +4.250000E-004 seconds

SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=0:1,
phOutput=&00007fff3657bbc4 )
    ---> Time elapsed - +1.303675E+000 seconds
```

```
SQLAllocHandle( phOutput=0:1 )
    <--- SQL_SUCCESS   Time elapsed - +1.920000E-004 seconds

SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=0:1,
phOutput=&00007fff3657bbc0 )
    ---> Time elapsed - +1.154550E+000 seconds

SQLAllocHandle( phOutput=0:1 )
    <--- SQL_SUCCESS   Time elapsed - +1.320000E-004 seconds
```

Usage notes:

You must enable CLI TRACE to use the **TraceAPIList** keyword.

TraceAPIList! CLI/ODBC configuration keyword

Specifies what APIs not to trace using the CLI trace facility. If you do not set the **TraceAPIList!** keyword, all APIs are traced.

To set the **TraceAPIList!** keyword in thedb2cli.ini file, issue the following command:

```
"db2 update cli cfg for section common using 'TRACEAPILIST!'API ID,API ID..."
```

where *API ID* is an integer that corresponds to the name of a CLI API. You can find the mapping between API names and IDs in the /sqllib/include/sqlcli1.h file, as shown in the following example:

```
#define  SQL_API_SQLALLOCHANDLE        1001
#define  SQL_API_SQLFREEHANDLE         1006
#define  SQL_API_SQLCLOSECURSOR        1003
#define  SQL_API_SQLENDTRAN            1005
#define  SQL_API_SQLCOLATTRIBUTE       6
#define  SQL_API_SQLGETSTMTATTR        1014
#define  SQL_API_SQLGETCONNECTATTR     1007
```

Note: To run this command, you must use double quotation marks around the full command and single quotation marks around the **TRACEAPILIST!** keyword.

The /sqllib/samples/cli/db2conn.c sample program depicts the behavior of different CLI connect APIs: SQLConnect, SQLDriverConnect, and SQLBrowseConnect. If you do not set the **TraceAPIList!** keyword, the CLI trace that is generated is similar to the following example:

```
[ Process: 4453, Thread: 47717036514016 ]
[ Date & Time:          06/26/2009 03:14:51.158736 ]
[ Product:              QDB2/LINUXX8664 DB2 v9.7.0.1 ]
[ Level Identifier:     08020107 ]
[ CLI Driver Version:   09.02.0000 ]
[ Informational Tokens: "DB2 v9.7.0.1","n090609","LINUXAMD6497","Fixpack 1" ]
[ Install Path:
/view/mayprasa_db2_v97fp1_linuxamd64_n090609_trc42/vbs/INST ]
[ db2cli.ini Location:  /home/mayprasa/sqllib/cfg/db2cli.ini ]
[ CLI Driver Type:      IBM DB2 Application Runtime Client ]



SQLAllocHandle( fHandleType=SQL_HANDLE_ENV, hInput=0:0,
phOutput=&00007fffb229a990 )
    ---> Time elapsed - 0 seconds

SQLAllocHandle( phOutput=0:1 )
    <--- SQL_SUCCESS   Time elapsed - +2.830000E-004 seconds

SQLSetEnvAttr( hEnv=0:1, fAttribute=SQL_ATTR_ODBC_VERSION, vParam=3, cbParam=0 )
```

```
    ---> Time elapsed - +4.300000E-005 seconds

SQLSetEnvAttr( )
    <--- SQL_SUCCESS   Time elapsed - +1.800000E-005 seconds

SQLAllocHandle( fHandleType=SQL_HANDLE_DBC, hInput=0:1,
phOutput=&00007fffb229a8e4 )
    ---> Time elapsed - +3.600000E-005 seconds

SQLAllocHandle( phOutput=0:1 )
    <--- SQL_SUCCESS   Time elapsed - +4.480000E-004 seconds

SQLConnect( hDbc=0:1, szDSN="sample", cbDSN=-3, szUID="", cbUID=-3,
szAuthStr="",
   cbAuthStr=-3 )
    ---> Time elapsed - +2.000000E-005 seconds
( DBMS NAME="DB2/LINUXX8664", Version="09.07.0001", Fixpack="0x28020107" )
( Application Codepage=819, Database  Codepage=1208, Database XML Codepage=1208,
   Char Send/Recv Codepage=819, Graphic Send/Recv Codepage=1200,
XML Send/Recv Codepage=1208 )

SQLConnect( )
    <--- SQL_SUCCESS   Time elapsed - +1.242704E+000 seconds
( DSN=""SAMPLE"" )
( UID=" " )
( PWD="" )
```

To prevent the SQLAllocHandle API from being traced, issue the following command:

```
"db2 update cli cfg for section common using 'TRACEAPILIST!' 1001"
```

where 1001 is the ID for **SQLAllocHandle** API. The CLI trace that is generated is similar to the following example:

```
[ Process: 5442, Thread: 47530732661472 ]
[ Date & Time:          06/26/2009 05:11:10.794067 ]
[ Product:              QDB2/LINUXX8664 DB2 v9.7.0.1 ]
[ Level Identifier:     08020107 ]
[ CLI Driver Version:   09.02.0000 ]
[ Informational Tokens: "DB2 v9.7.0.1","n090609","LINUXAMD6497","Fixpack 1" ]
[ Install Path:
/view/mayprasa_db2_v97fp1_linuxamd64_n090609_trc42/vbs/INST ]
[ db2cli.ini Location:  /home/mayprasa/sqllib/cfg/db2cli.ini ]
[ CLI Driver Type:      IBM DB2 Application Runtime Client ]



SQLSetEnvAttr( hEnv=0:1, fAttribute=SQL_ATTR_ODBC_VERSION, vParam=3, cbParam=0 )
    ---> Time elapsed - 0 seconds

SQLSetEnvAttr( )
    <--- SQL_SUCCESS   Time elapsed - +2.200000E-005 seconds

SQLConnect( hDbc=0:1, szDSN="sample", cbDSN=-3, szUID="", cbUID=-3, szAuthStr="",
   cbAuthStr=-3 )
    ---> Time elapsed - +4.850000E-004 seconds
( DBMS NAME="DB2/LINUXX8664", Version="09.07.0001", Fixpack="0x28020107" )
( Application Codepage=819, Database  Codepage=1208, Database XML Codepage=1208,
   Char Send/Recv Codepage=819, Graphic Send/Recv Codepage=1200,
XML Send/Recv Codepage=1208 )

SQLConnect( )
    <--- SQL_SUCCESS   Time elapsed - +1.156099E+000 seconds
( DSN=""SAMPLE"" )
( UID=" " )
( PWD="" )
```

```
SQLDisconnect( hDbc=0:1 )
    ---> Time elapsed - +4.400000E-005 seconds

SQLDisconnect( )
    <--- SQL_SUCCESS   Time elapsed - +1.197430E-001 seconds
```

Usage notes:

You must enable CLI TRACE to use the **TraceAPIList!** keyword.

TraceComm CLI/ODBC configuration keyword

Specifies whether information about each network request is included in the trace file.

db2cli.ini keyword syntax:

TraceComm = **0** | **1**

Default setting:

0 - No network request information is captured.

Only applicable when:

the CLI/ODBC Trace option is turned on.

Usage notes:

When TraceComm is set on (1) then the following information about each network request will be included in the trace file:

- which CLI functions are processed completely on the client and which CLI functions involve communication with the server
- the number of bytes sent and received in each communication with the server
- the time spent communicating data between the client and server

This option is only used when the Trace CLI/ODBC option is turned on.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TraceErrImmediate CLI/ODBC configuration keyword

Specifies whether diagnostic records are written to the CLI/ODBC trace when records are generated.

db2cli.ini keyword syntax:

TraceErrImmediate = **0** | **1**

Default setting:

Diagnostic records are only written to the trace file when `SQLGetDiagField()` or `SQLGetDiagRec()` is called; or "Unretrieved Error Message" is written to the trace file for handles which had diagnostic records that were left unretreived.

Only applicable when:

the CLI/ODBC Trace option is turned on.

Usage notes:

Setting TraceErrImmediate=1 helps in determining when errors occur during application execution by writing diagnostic records to the CLI/ODBC trace file at the time the records are generated. This is especially useful for applications that do not retrieve diagnostic information using `SQLGetDiagField()` and `SQLGetDiagRec()`, because the diagnostic records that were generated on a handle will be lost if they are not retrieved or written to the trace file before the next function is called on the handle.

If TraceErrImmediate=0 (the default setting), then diagnostic records will only be written to the trace file if an application calls `SQLGetDiagField()` or `SQLGetDiagRec()` to retrieve diagnostic information. If the application does not retrieve diagnostic information through function calls and this keyword is set to 0, then the "Unretrieved Error Message" entry will be written to the trace file if a diagnostic record exists, when a function is next called on the handle.

This option is only used when the Trace CLI/ODBC option is turned on.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TraceFileName CLI/ODBC configuration keyword

Specifies a file to which all CLI/ODBC trace information is written.

db2cli.ini keyword syntax:
TraceFileName = < **fully qualified file name** >

Default setting:
None

Only applicable when:
the Trace option is turned on.

Usage notes:

If the file specified does not exist, then it will be created; otherwise, the new trace information will be appended to the end of the file. However, the path the file is expected in must exist.

If the filename given is invalid or if the file cannot be created or written to, no trace will occur and no error message will be returned.

This option is only used when the Trace option is turned on. This will be done automatically when you set this option in the CLI/ODBC Configuration utility.

The TracePathName option will be ignored if this option is set.

CLI trace should only be used for debugging purposes. It will slow down the execution of the CLI/ODBC driver, and the trace information can grow quite large if it is left on for extended periods of time.

The TraceFileName keyword option should not be used with multi-process or multithreaded applications as the trace output for all threads or processes will be written to the same log file, and the output for each thread or process will be difficult to decipher. Furthermore, semaphores are used to control access to the shared trace file which could change the behavior of multithreaded applications. There is no default DB2 CLI trace output log file name.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2 databases.)

TraceFlush CLI/ODBC configuration keyword

Forces a write to disk after n CLI/ODBC trace entries.

db2cli.ini keyword syntax:
TraceFlush = **0** | **positive integer**

Default setting:
Do not write after every entry.

Only applicable when:
the CLI/ODBC Trace option is turned on.

Usage notes:

TraceFlush specifies how often trace information is written to the CLI trace file. By default, TraceFlush is set to 0 and each DB2 CLI trace file is kept open until the traced application or thread terminates normally. If the application terminates abnormally, some trace information that was not written to the trace log file may be lost.

Set this keyword to a positive integer to force the CLI driver to close and re-open the appropriate trace file after the specified number of trace entries. The smaller the value of the TraceFlush keyword, the greater the impact CLI tracing has on the performance of the application. Setting TraceFlush=1 has the most impact on performance, but will ensure that each entry is written to disk before the application continues to the next statement.

This option is only used when the Trace CLI/ODBC option is turned on.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TraceFlushOnError CLI/ODBC configuration keyword

Specifies whether all CLI/ODBC trace entries are written to disk when an error occurs.

db2cli.ini keyword syntax:
TraceFlushOnError = **0** | **1**

Default setting:
Do not write CLI/ODBC trace entries as soon as an error occurs.

Only applicable when:
the CLI/ODBC Trace option is turned on.

Usage notes:

Setting TraceFlushOnError=1 forces the CLI driver to close and re-open the trace file each time an error is encountered. If TraceFlushOnError is left at its default value of 0, then trace file will only be closed when the application terminates normally or the interval specified by the TraceFlush keyword is reached. If the application process were to terminate abnormally when TraceFlushOnError=0, then

valuable trace information may be lost. Setting TraceFlushOnError=1 may impact performance, but will ensure that trace entries associated with errors are written to disk.

This option is only used when the Trace CLI/ODBC option is turned on.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TraceLocks CLI/ODBC configuration keyword

Only trace lock timeouts in the CLI/ODBC trace.

db2cli.ini keyword syntax:
TraceLocks = **0** | **1**

Default setting:
Trace information is not limited to only lock timeouts.

Only applicable when:
the Trace option is turned on.

Usage notes:

When TraceLocks is set to 1, lock timeouts will be recorded in the trace file.

This option is only used when the CLI/ODBC TRACE option is turned on.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TracePIDList CLI/ODBC configuration keyword

Restricts the process IDs for which the CLI/ODBC trace will be enabled.

db2cli.ini keyword syntax:
TracePIDList = **<no value specified>** | **<comma-delimited list of process IDs>**

Default setting:
All of the process IDs will be traced when the CLI/ODBC trace is run.

Usage notes:

Use this keyword for applications that create many processes. Capturing the CLI/ODBC trace for such applications can generate many trace files. By using this keyword you can collect the trace of specific problematic processes of applications.

If no value is specified for this keyword, all process IDs will be traced. Otherwise, specify a comma-delimited list of process IDs which you want to be traced when the CLI/ODBC trace runs.

The TraceRefreshInterval keyword must be set to some value before initializing your application, otherwise, the TracePIDList keyword will not take effect.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2 databases.)

To use the TracePIDList keyword:

1. Ensure the Trace CLI/ODBC keyword is set to zero or is not specified in the db2cli.ini file.
2. Add the TraceRefreshInterval CLI/ODBC keyword to the Common section of the db2cli.ini file as follows:

   ```
   [COMMON]
   TraceRefreshInterval=<some positive integer>
   ```
3. Start your application.
4. Using an operating system command such as **ps** (on a UNIX and Linux-based operating systems), determine the process IDs of the processes that you want to collect the CLI/ODBC trace for.
5. Turn CLI/ODBC tracing on and add the process IDs identified to the Common section of the db2cli.ini file by including the following keywords:

   ```
   [COMMON]
   Trace=1
   TracePathName=<fully-qualified subdirectory name>
   TracePIDList=<comma-delimited list of process IDs>
   ```

CLI/ODBC traces containing information of the process IDs specified will be located in the directory specified by the TracePathName keyword.. You might also see extra empty files that can be ignored.

TracePIDTID CLI/ODBC configuration keyword

Captures the process ID and thread ID for each item being traced.

db2cli.ini keyword syntax:

TracePIDTID = **0** | **1**

Default setting:

The process ID and thread ID for the trace entries are not captured.

Only applicable when:

the Trace option is turned on.

Usage notes:

When TracePIDTID is set to 1, the process ID and thread ID for each captured item will be recorded in the trace file. This effect is helpful when the Trace keyword is enabled and multiple applications are executing. This is because Trace writes trace information for all executing applications to a single file. Enabling TracePIDTID differentiates the recorded information by process and thread.

This option is only used when the CLI/ODBC Trace option is turned on.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TracePathName CLI/ODBC configuration keyword

Specifies the subdirectory to be used to store individual CLI/ODBC trace files.

db2cli.ini keyword syntax:

TracePathName = < **fully qualified subdirectory name** >

Default setting:

None

Only applicable when:
the Trace option is turned on.

Not applicable when:
the TraceFileName option is turned on.

Usage notes:

Each thread or process that uses the same DLL or shared library will have a separate CLI/ODBC trace file created in the specified directory. A concatenation of the application process ID and the thread sequence number is automatically used to name trace files.

No trace will occur, and no error message will be returned, if the subdirectory given is invalid or if it cannot be written to.

This option is only used when the Trace option is turned on. This will be done automatically when you set this option in the CLI/ODBC Configuration utility.

It will be ignored if the CLI/ODBC option TraceFileName is used.

CLI trace should only be used for debugging purposes. It will slow down the execution of the CLI/ODBC driver, and the trace information can grow quite large if it is left on for extended periods of time.

If both TraceFileName and TracePathName are specified, the TraceFileName keyword takes precedence and TracePathName will be ignored.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TraceRefreshInterval CLI/ODBC configuration keyword

Sets the interval (in seconds) at which the **Trace** and **TracePIDList** keywords are read from the Common section of the `db2cli.ini` file.

db2cli.ini keyword syntax:
TraceRefreshInterval = **0** | **positive integer**

Default setting:
The **Trace** and **TracePIDList** keywords will only be read from the `db2cli.ini` file when the application is initialized.

Usage notes:

Setting this keyword before an application is initialized allows you to dynamically turn off the CLI/ODBC trace within *n* seconds.

Note: Setting **TraceRefreshInterval** while the application is running will have no effect. For this keyword to take effect, it must be set before the application is initialized.

Only the **Trace** and **TracePIDList** keywords will be refreshed from the `db2cli.ini` file if this keyword is set. No other CLI or ODBC configuration keywords will be reread.

When **TraceRefreshInterval** is set to a nonzero positive integer value, a thread is spawned to monitor the `db2cli.ini`. In this situation, the

applications connected to the database need to be multithread safe; otherwise the application might behave in an unexpected manner.

This keyword is contained in the Common section of the initialization file and therefore applies to all connections to DB2.

Note: This CLI keyword is ignored if it is used inside a stored procedure or routine that uses CLI API calls.

TraceStmtOnly CLI/ODBC configuration keyword

Only trace dynamic SQL statements in the CLI/ODBC trace.

db2cli.ini keyword syntax:
TraceStmtOnly = **0** | **1**

Default setting:
Trace information is not limited to only dynamic SQL statements.

Only applicable when:
the Trace option is turned on.

Usage notes:

When TraceStmtOnly is set to 1, only dynamic SQL statements will be recorded in the trace file.

This option is only used when the CLI/ODBC Trace option is turned on.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TraceTime CLI/ODBC configuration keyword

Captures elapsed time counters in the trace file.

db2cli.ini keyword syntax:
TraceTime = **1** | **0**

Default setting:
Elapsed time counters are included in the trace file.

Only applicable when:
the Trace option is turned on.

Usage notes:

When TraceTime is set to 1, elapsed time counters will be captured in the trace file. For example:

```
SQLPrepare( hStmt=1:1, pszSqlStr="SELECT * FROM ORG", cbSqlStr=-3 )
  ---> Time elapsed - +6.785751E+000 seconds ( StmtOut="SELECT * FROM ORG" )
SQLPrepare( )
  <--- SQL_SUCCESS Time elapsed - +2.527400E-002 seconds
```

Turn TraceTime off, by setting it to 0, to improve performance or to generate smaller trace files. For example:

```
SQLPrepare( hStmt=1:1, pszSqlStr="SELECT * FROM ORG", cbSqlStr=-3 )
( StmtOut="SELECT * FROM ORG" )
SQLPrepare( )
  <—— SQL_SUCCESS
```

This option is only used when the CLI/ODBC Trace option is turned on.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

TraceTimestamp CLI/ODBC configuration keyword

Specifies what type of timestamp information (if any) is recorded in the CLI/ODBC trace.

db2cli.ini keyword syntax:

TraceTimestamp = **0** | **1** | **2** | **3**

Default setting:

No timestamp information is written to the trace file.

Only applicable when:

the Trace option is turned on.

Usage notes:

Setting TraceTimeStamp to a value other than the default of 0 means the current timestamp or absolute execution time is added to the beginning of each line of trace information as it is being written to the DB2 CLI trace file. The following settings indicate what type of timestamp information is captured in the trace file:

- 0 = no timestamp information
- 1 = processor ticks and ISO timestamp (absolute execution time in seconds and milliseconds, followed by a timestamp)
- 2 = processor ticks (absolute execution time in seconds and milliseconds)
- 3 = ISO timestamp

This option is only used when the CLI/ODBC Trace option is turned on.

(This option is contained in the Common section of the initialization file and therefore applies to all connections to DB2.)

Trusted_Connection CLI/ODBC configuration keyword

Specifies whether a connection made with the current authenticated user is allowed.

Syntax:

Trusted_Connection=Yes

Note: This keyword will have no effect if set in the `db2cli.ini` file. It should instead be provided in the connection string to SQLDriverConnect().

Default setting:

CLI uses the user ID and password information provided in the connection string to `SQLDriverConnect()`, not the current authenticated user.

Usage notes:

CLI applications that connect to a database will typically connect using the function SQLDriverConnect(). One of the input arguments for this function is the *DriverCompletion* value, which determines when a dialog will be opened. The valid values of the *DriverCompletion* argument are:

- SQL_DRIVER_PROMPT: A dialog is always initiated.
- SQL_DRIVER_COMPLETE: A dialog is only initiated if there is insufficient information in the connection string.
- SQL_DRIVER_COMPLETE_REQUIRED: A dialog is only initiated if there is insufficient information in the connection string. Only mandatory information is requested. The user is prompted for required information only.
- SQL_DRIVER_NOPROMPT: The user is not prompted for any information. A connection is attempted with the information contained in the connection string. If there is not enough information, SQL_ERROR is returned.

Note: More details on the *DriverCompletion* argument can be found in the documentation for SQLDriverConnect().

Some applications, for example, those in a Kerberos environment, might require that a user be able to connect to a DB2 server without providing a user ID or password. If the application uses the SQL_DRIVER_NO_PROMPT option on the SQLDriverConnect() call, the connection will be attempted without the user authentication. This keyword is then not required.

In the case where a third party application is involved and the prompt level used by the application is something other than SQL_DRIVER_NO_PROMPT, CLI will open a dialog to request the missing information from the user. Setting **Trusted_Connection** to Yes, by providing it to the input connection string for SQLDriverConnect() ("Trusted_Connection=Yes"), causes CLI to ignore any user ID or password string (including blank strings) from the connection string and ignore the prompt level of the connection function. CLI will use the current authenticated user to attempt the connection to the database. If the connection attempt fails, the user will be prompted for the user ID and password.

This keyword is used only in the connection string for SQLDriverConnect(); setting it in the db2cli.ini file will have no effect.

TxnIsolation CLI/ODBC configuration keyword

Sets the default isolation level.

db2cli.ini keyword syntax:

```
TxnIsolation = ReadUncommitted | ReadCommitted | RepeatableRead | Serializable | NoCommit | 1 | 2 | 4 | 8 | 32
```

Default setting:

2 or ReadCommitted (Cursor Stability)

Only applicable when:

the default isolation level is used. This keyword will have no effect if the application has specifically set the isolation level.

Equivalent statement attribute:

SQL_ATTR_TXN_ISOLATION

Usage notes:

Sets the isolation level to:
- 1 = SQL_TXN_READ_UNCOMMITTED - Read uncommitted (Uncommitted read)
- 2 = SQL_TXN_READ_COMMITTED (default) - Read committed (Cursor stability)
- 4 = SQL_TXN_REPEATABLE_READ - Repeatable read (Read stability)
- 8 = SQL_TXN_SERIALIZABLE - Serializable (Repeatable read)
- 32 = SQL_TXN_NOCOMMIT - (No commit, DB2 Universal Database for AS/400® only; this setting is similar to autocommit).

The words in parentheses are the IBM terminology for the equivalent SQL92 isolation levels. Note that *no commit* is not an SQL92 isolation level and is supported on IBM DB2 for IBM i only.

Table 166. Supported isolation levels

Isolation level	Keyword	SQL92	IBM terminology
1	SQL_TXN_READ_UNCOMMITTED	Read uncommitted	Uncommitted read
2	SQL_TXN_READ_COMMITTED (default)	Read committed	Cursor stability
4	SQL_TXN_REPEATABLE_READ	Repeatable read	Read stability
8	SQL_TXN_SERIALIZABLE	Serializable	Repeatable read
32	SQL_TXN_NOCOMMIT	Not an SQL92 isolation level	No commit

You can use the listed textual values to set the *TxnIsolation* keyword in the `db2cli.ini` file:
- ReadUncommitted
- ReadCommitted
- RepeatableRead
- Serializable
- NoCommit

If you use a text value that is not in the list, the value is ignored and *TxnIsolation* is set to the default value.

This keyword is only applicable if you use the default isolation level. If the application has explicitly set the isolation level for a connection or statement handle, this keyword setting is ignored.

UID CLI/ODBC configuration keyword

Defines a default user ID.

db2cli.ini keyword syntax:
UID = *userid*

Default setting:
None

Usage notes:

The specified *userid* value is used if a userid is not provided by the application at connect time.

Underscore CLI/ODBC configuration keyword

Specifies whether the underscore character '_' is treated as a wildcard.

db2cli.ini keyword syntax:

Underscore = **0** | **1**

Default setting:

The underscore character matches any single character or no character.

Usage notes:

This keyword specifies if the underscore character '_' will be recognized as a wildcard or only as the underscore character. The possible settings are as follows:

- 0 - The underscore character is treated only as the underscore character.
- 1 - The underscore character is treated as a wildcard that matches any single character, including no character.

Setting Underscore to 0 can improve performance when there are database objects with names that contain underscores.

This keyword applies only to the following catalog functions that accept search patterns as arguments:

- `SQLColumnPrivileges()`
- `SQLColumns()`
- `SQLProcedureColumns()`
- `SQLProcedures()`
- `SQLTablePrivileges()`
- `SQLTables()`

Note that catalog functions may only accept search patterns on particular arguments. Refer to the documentation of the specific function for details.

UseOldStpCall CLI/ODBC configuration keyword

Controls how cataloged procedures are invoked.

db2cli.ini keyword syntax:

UseOldStpCall = **0** | **1**

Default setting:

Invokes procedures using the new CALL method where GRANT EXECUTE must be granted on the procedure.

Usage notes:

Prior to DB2 Universal Database Version 8, the invoker of a procedure had to have EXECUTE privilege on any package invoked from the procedure. Now, the invoker must have EXECUTE privilege on the procedure and only the definer of the procedure has to have EXECUTE privilege on any required packages.

This keyword controls which method is used to invoke the procedure. Setting UseOldStpCall on causes the procedure to be invoked using the deprecated sqleproc() API when the precompiler fails to resolve a procedure on a CALL statement. Turning this keyword off will invoke procedures where GRANT EXECUTE must be granted on the procedure.

UseServerMsgSP CLI/ODBC configuration keyword

Specifies whether a stored procedure is called to retrieve message text when connected to DB2 for z/OS servers.

db2cli.ini keyword syntax:

UseServerMsgSP = **0** | **1**

Default setting:

CLI does not use the server stored procedure to return messages, but uses the local message files.

Equivalent connection attribute:

SQL_ATTR_SERVER_MSGTXT_SP

Usage notes:

CLI calls the stored procedure indicated by the SQL_ATTR_SERVER_MSGTXT_SP connection attribute. If this attribute is not set, CLIcalls the SYSIBM.SQLCAMESSAGE stored procedure. If this attribute is set to DSNACCMG, CLI calls DSNACCMG when connected to DB2 for z/OS Version 7 servers and calls SYSIBM.SQLCAMESSAGE when connected to DB2 for z/OS Version 8 or later.

Applications using this keyword should also set the "ServerMsgMask CLI/ODBC configuration keyword" on page 398 to indicate when CLI should call this procedure to retrieve the message information from the server. If the "ServerMsgMask CLI/ODBC configuration keyword" on page 398 is not set, then the default is to check the local message files first. See the "ServerMsgMask CLI/ODBC configuration keyword" on page 398 for more details on the options available.

DSNACCMG has been deprecated in DB2 for z/OS Version 9 and might be removed in a future release. If SQL_ATTR_SERVER_MSGTXT_SP is set to DSNACCMG, set this attribute to a different store procedure to retrieve messages text. Alternatively, use local message files or use the ServerMsgTextSP configuration keyword.

ServerMsgTextSP CLI/ODBC configuration keyword

Specifies which stored procedure is used to retrieve message text from DB2 for z/OS.

db2cli.ini keyword syntax:

ServerMsgTextSP = **stored procedure name**

Default setting:

CLI does not use the server stored procedure to return messages, but uses the local message files.

Equivalent connection attribute:

SQL_ATTR_SERVER_MSGTXT_SP

Usage notes:

Applications using this keyword should also set the "ServerMsgMask CLI/ODBC configuration keyword" on page 398 to indicate when CLI should call this procedure to retrieve the message information from the server. If the "ServerMsgMask CLI/ODBC configuration keyword" on page 398 is not set, then the default is to check the local message files first. See the "ServerMsgMask CLI/ODBC configuration keyword" on page 398 for more details on the options available.

The difference between **ServerMsgTextSP** and "UseServerMsgSP CLI/ODBC configuration keyword" on page 424 is that **UseServerMsgSP** can be turned on and off to call the procedure specified in the SQL_ATTR_SERVER_MSGTXT_SP connection attribute, while **ServerMsgTextSP** needs to have the procedure explicitly specified.

WarningList CLI/ODBC configuration keyword

Specifies which errors to downgrade to warnings.

db2cli.ini keyword syntax:

WarningList = " 'xxxxx', 'yyyyy', ..."

Default setting:

Do not downgrade any SQLSTATEs.

Usage notes:

Any number of SQLSTATEs returned as errors can be downgraded to warnings. Each must be delimited with single quotation mark, separated by commas, and in uppercase. The entire string must also be enclosed in double quotation marks. For example:

```
WarningList=" '01S02', 'HY090' "
```

XMLDeclaration CLI/ODBC configuration keyword

Controls the generation of an XML declaration when XML data is implicitly serialized to an application variable.

db2cli.ini keyword syntax:

XMLDeclaration = **non-negative integer < 7** | **7**

Default setting:

A BOM and an XML declaration containing the XML version and encoding attribute are generated during implicit serialization.

Usage notes:

The XMLDeclaration keyword controls which elements of an XML declaration are prepended to an application buffer when XML data is implicitly serialized to an application buffer. This setting does not affect the result of the XMLSERIALIZE function.

The following values represent components to be generated during implicit serialization. Set this keyword by adding together the value of each component required.

0 No declarations or byte order marks (BOMs) are added to the output buffer.

1 A byte order mark (BOM) in the appropriate endianness is prepended to the output buffer if the target encoding is UTF-16 or UTF-32. (Although a UTF-8 BOM exists, the database server does not generate it, even if the target encoding is UTF-8.)

2 A minimal XML declaration is generated, containing only the XML version.

4 An encoding attribute that identifies the target encoding is added to any

generated XML declaration. Therefore, this setting only has effect when the setting of 2 is also included when computing the value of this keyword.

For example, if you wanted a BOM and minimal XML declaration (without an encoding attribute) to be generated during implicit serialization, you would set `XMLDeclaration = 3`, where 3 is the sum of 1 (the value to indicate generation of a BOM) and 2 (the value to indicate generation of a minimal XML declaration).

To prevent any declarations or BOM from being generated, set XMLDeclaration as follows: `XMLDeclaration = 0`.

Chapter 4. Environment, connection, and statement attributes in CLI applications

Environments, connections, and statements each have a defined set of attributes (or options). All attributes can be queried by the application, but only some attributes can be changed from their default values. By changing attribute values, the application can change the behavior of CLI.

An environment handle has attributes which affect the behavior of CLI functions under that environment. The application can specify the value of an attribute by calling `SQLSetEnvAttr()` and can obtain the current attribute value by calling `SQLGetEnvAttr()`. `SQLSetEnvAttr()` can only be called before any connection handles have been allocated for the environment handle. For details on each environment attribute, see the list of CLI environment attributes.

A connection handle has attributes which affect the behavior of CLI functions under that connection. Of the attributes that can be changed:

- Some can be set any time after the connection handle is allocated.
- Some can be set only before the actual connection has been established.
- Some can be set any time after the connection has been established.
- Some can be set after the connection has been established, but only while there are no outstanding transactions or open cursors.

The application can change the value of connection attributes by calling `SQLSetConnectAttr()` and can obtain the current value of an attribute by calling `SQLGetConnectAttr()`. An example of a connection attribute which can be set any time after a handle is allocated is the auto-commit option SQL_ATTR_AUTOCOMMIT. For details on each connection attribute, see the list of CLI connection attributes.

A statement handle has attributes which affect the behavior of CLI functions executed using that statement handle. Of the statement attributes that can be changed:

- Some attributes can be set, but currently are limited to only one specific value.
- Some attributes can be set any time after the statement handle has been allocated.
- Some attributes can only be set if there is no open cursor on that statement handle.

The application can specify the value of any statement attribute that can be set by calling `SQLSetStmtAttr()` and can obtain the current value of an attribute by calling `SQLGetStmtAttr()`. For details on each statement attribute, see the list of CLI statement attributes.

The `SQLSetConnectAttr()` function cannot be used to set statement attributes. This was supported in versions of CLI before version 5.

Many applications just use the default attribute settings; however, there might be situations where some of these defaults are not suitable for a particular user of the application. Some default values can be changed by setting the CLI/ODBC configuration keywords. CLI provides end users with two methods of setting some

configuration keywords. The first method is to specify the keyword and its new default attribute value(s) in the connection string input to the SQLDriverConnect() and SQLBrowseConnect() functions. The second method involves the specification of the new default attribute value(s) in a CLI initialization file using CLI/ODBC configuration keywords.

The CLI initialization file can be used to change default values for all CLI applications on that workstation. This might be the end user's only means of changing the defaults if the application does not provide a means for the user to provide default attribute values in the SQLDriverConnect() connection string. Default attribute values that are specified on SQLDriverConnect() override the values in the CLI initialization file for that particular connection.

The mechanisms for changing defaults are intended for end user tuning; application developers must use the appropriate set-attribute function. If an application does call a set-attribute or option function with a value different from the initialization file or the connection string specification, then the initial default value is overridden and the new value takes effect.

The figure-1 shows the addition of the attribute functions to the basic connect scenario.

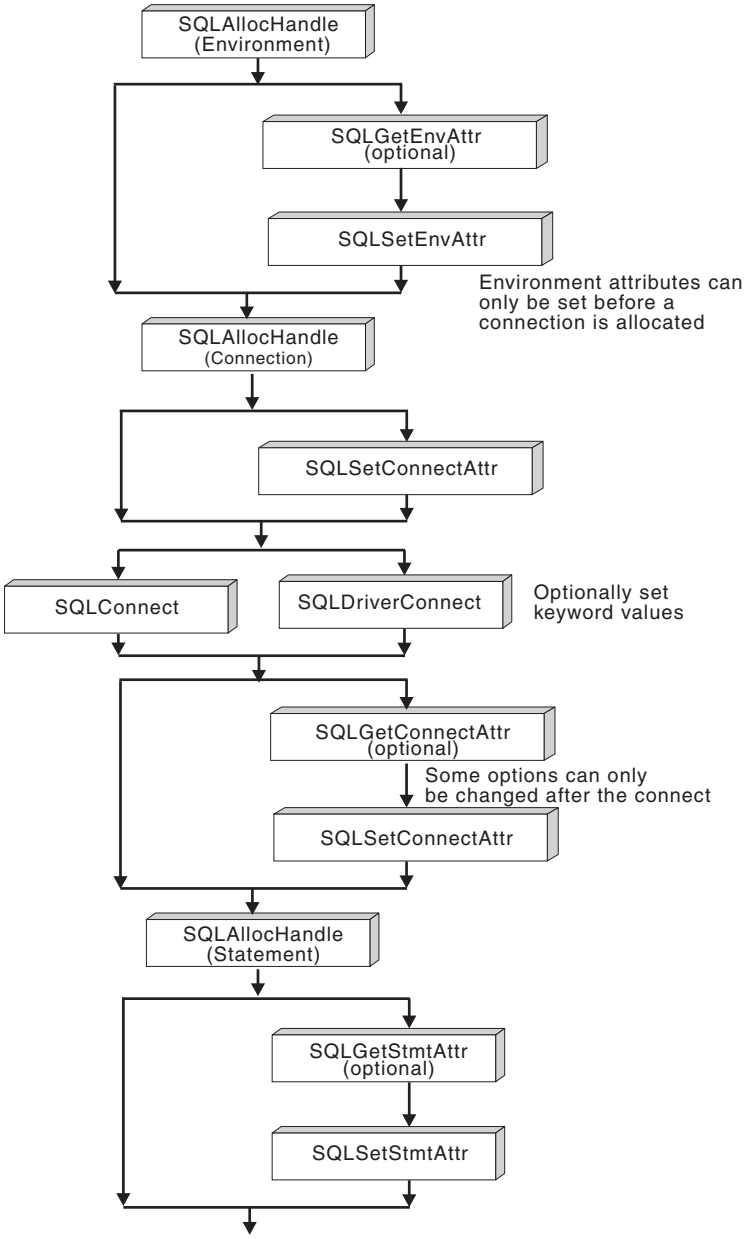


Figure 1. Setting and retrieving attributes (options)

Environment attributes (CLI) list

CLI environment attributes that can be set using the `SQLSetEnvAttr()`.

ODBC does not support setting driver-specific environment attributes using `SQLSetEnvAttr()`. Only CLI applications can set the CLI-specific environment attributes using this function.

SQL_ATTR_CONNECTION_POOLING
This attribute was deprecated in DB2 UDB for Linux, UNIX, and Windows Version 8.

This attribute is not supported when accessing the Informix database server.

SQL_ATTR_CONNECTTYPE

This attribute replaces the SQL_CONNECTTYPE attribute. A 32-bit integer value that specifies whether this application is to operate in a coordinated or uncoordinated distributed environment. The possible values are:

- **SQL_CONCURRENT_TRANS**: The application can have concurrent multiple connections to any one database or to multiple databases. Each connection has its own commit scope. No effort is made to enforce the coordination of the transaction. If an application issues a commit by using the environment handle on `SQLEndTran()` and not all of the connections commit successfully, the application is responsible for recovery. This is the default.
- **SQL_COORDINATED_TRANS**: The application can coordinate commit and rollbacks among multiple database connections. This option setting corresponds to the specification of the Type 2 CONNECT in embedded SQL. In contrast to the SQL_CONCURRENT_TRANS setting that was previously described, the application is permitted only one open connection per database.

  Note: This connection type results in the default for the SQL_ATTR_AUTOCOMMIT connection option to be SQL_AUTOCOMMIT_OFF.

If you change this attribute from the default, you must set it before any connections are established on the environment handle.

Application typically set this attribute as an environment attribute with a call to `SQLSetEnvAttr()` function. The `SQLSetEnvAttr()` function is called as soon as the environment handle is allocated. However, because ODBC applications cannot access `SQLSetEnvAttr()` function, ODBC applications must set this attribute using `SQLSetConnectAttr()` function after each connection handle is allocated, but before any connections are established.

All connections on an environment handle must have the same SQL_ATTR_CONNECTTYPE setting. An environment cannot have both concurrent and coordinated connections. The type of the first connection determines the type of all subsequent connections. `SQLSetEnvAttr()` returns an error if an application attempts to change the connection type while there is an active connection.

You can also set the default connect type by using the “ConnectType CLI/ODBC configuration keyword” on page 344.

The SQL_ATTR_CONNECTTYPE attribute is an IBM defined extension.

SQL_ATTR_CP_MATCH

This attribute was deprecated in DB2 database version 8.

This attribute is not supported when accessing the Informix database server.

SQL_ATTR_DIAGLEVEL

Description

A 32-bit integer value which represents the diagnostic level. This is equivalent to the database manager `DIAGLEVEL`parameter.

Values

Valid values are: 0, 1, 2, 3, or 4. (The default value is 3.)

Usage notes

You must set this attribute before any connection handles are created.

SQL_ATTR_DIAGPATH

Description

A pointer to a null-terminated character string that contains the name of the directory where diagnostic data is to be placed. This is equivalent to the database manager `DIAGPATH` parameter.

Values

The default value is the `db2dump` directory on UNIX and Linux operating systems, and the `db2` directory on Windows operating systems.

Usage notes

You must set this attribute before any connection handles are created.

SQL_ATTR_INFO_ACCTSTR

Description

A pointer to a null-terminated character string that is used to identify the client accounting string that is sent to the data server when using DB2 Connect or DB2 database products for Linux, UNIX, and Windows.

Values

When you set the value, some servers might not be able to handle the entire length that is provided and might truncate the value. DB2 for z/OS and DB2 Universal Database for z/OS and OS/390 servers support up to 200 characters. To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

The SQL_ATTR_INFO_ACCTSTR attribute is an IBM defined extension.

SQL_ATTR_INFO_APPLNAME

Description

A pointer to a null-terminated character string that is used to identify the client application name that is sent to the data server when using DB2 Connect or DB2 database products for Linux, UNIX, and Windows.

Values

When you set the value, some servers might not be able to handle the entire length provided and might truncate the value. DB2 for z/OS and DB2 Universal Database for z/OS and OS/390 servers support up to 32 characters. To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

The SQL_ATTR_INFO_APPLNAME attribute is an IBM defined extension.

SQL_ATTR_INFO_USERID

Description

A pointer to a null-terminated character string that is used to

identify the client user ID that is sent to the data server when using DB2 Connect or DB2 database products for Linux, UNIX, and Windows.

Values

When you set the value, some servers might not be able to handle the entire length provided and might truncate the value. DB2 for z/OS and DB2 Universal Database for z/OS and OS/390 servers support up to 16 characters. This user ID is not to be confused with the authentication user ID. This user ID is for identification purposes only, and is not used for any authorization. To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

The SQL_ATTR_INFO_USERID attribute is an IBM defined extension.

SQL_ATTR_INFO_WRKSTNNAME

Description

A pointer to a null-terminated character string that is used to identify the client workstation name that is sent to the data server when using DB2 Connect or DB2 database products for Linux, UNIX, and Windows.

Values

When you set the value, some servers might not be able to handle the entire length provided and might truncate the value. DB2 for z/OS and DB2 Universal Database for z/OS and OS/390 servers support up to 18 characters. To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

The SQL_ATTR_INFO_WRKSTNNAME attribute is an IBM defined extension.

SQL_ATTR_MAXCONN

This attribute was deprecated in DB2 Version 8.

This attribute is not supported when accessing the Informix database servers.

SQL_ATTR_NOTIFYLEVEL

Description

A 32-bit integer value that represents the notification level. This is equivalent to the database manager `NOTIFYLEVEL` parameter.

Values

Valid values are: 0, 1, 2, 3, or 4. (The default value is 3.)

Usage notes

You must set this attribute value before any connection handles are created.

This attribute is not supported when accessing the Informix database servers.

SQL_ATTR_ODBC_VERSION

Description

A 32-bit integer that determines whether certain functionality

exhibits ODBC 2.x (CLI v2) behavior or ODBC 3.0 (CLI v5) behavior. ODBC applications must set this environment attribute before calling any function that has an SQLHENV argument, or the call will return SQLSTATE HY010 (Function sequence error.).

Values

The listed values are used to set the value of this attribute:

- SQL_OV_ODBC3: Causes the listed ODBC 3.0 (CLI v5) behavior:
  - CLI returns and expects ODBC 3.0 (CLI v5) codes for date, time, and timestamp.
  - CLI returns ODBC 3.0 (CLI v5) SQLSTATE codes when `SQLError()`, `SQLGetDiagField()`, or `SQLGetDiagRec()` functions are called.
  - The *CatalogName* argument in a call to `SQLTables()` function accepts a search pattern.
- SQL_OV_ODBC2: Causes the listed ODBC 2.x (CLI v2) behavior:
  - CLI returns and expects ODBC 2.x (CLI v2) codes for date, time, and timestamp.
  - CLI returns ODBC 2.0 (CLI v2) SQLSTATE codes when `SQLError()`, `SQLGetDiagField()`, or `SQLGetDiagRec()` functions are called.
  - The *CatalogName* argument in a call to `SQLTables()` function does not accept a search pattern.
- SQL_OV_ODBC3_80: Causes the listed ODBC 3.0 (CLI v5) behavior:
  - CLI returns and expects ODBC 3.x codes for date, time, and timestamp.
  - CLI returns ODBC 3.x SQLSTATE codes when `SQLError()`, `SQLGetDiagField()`, or `SQLGetDiagRec()` functions are called.
  - The *CatalogName* argument in a call to `SQLTables()` function accepts a search pattern.

SQL_ATTR_OUTPUT_NTS

Description

A 32-bit integer value that controls the use of null-termination in output arguments.

Values

The possible values are:

- **SQL_TRUE**: CLI uses null termination to indicate the length of output character strings (default).
- **SQL_FALSE**: CLI does not use null termination in output character strings.

The CLI functions that are affected by this attribute are all of the functions that are called for the environment (and for any connections and statements that are allocated under the environment) that have character string parameters.

You can only set this attribute when there are no connection handles that are allocated under this environment.

SQL_ATTR_PROCESSCTL

Description

A 32-bit mask that sets process-level attributes, which affect all environments and connections for the process. You must set this attribute before the environment handle is allocated.

The call to SQLSetEnvAttr() must have the *EnvironmentHandle* argument set to SQL_NULL_HANDLE. The settings remain in effect for the duration of the process. Generally, use this attribute only for performance sensitive applications, where large numbers of CLI function calls are being made. Before setting any of these bits, ensure that the application, and any other libraries that the application calls, comply with the restrictions that are listed.

Values

You can combine the listed values to form a bit mask:

- SQL_PROCESSCTL_NOTHREAD - This bit indicates that the application does not use multiple threads, or if it does use multiple threads, guarantees that all DB2 calls are serialized by the application. If set, CLI does not make any system calls to serialize calls to CLI, and sets the DB2 context type to SQL_CTX_ORIGINAL.
- SQL_PROCESSCTL_NOFORK - This bit indicates that the application will never fork a child process. By default, CLI does not check to see if an application forks a child process. However, if the CheckForFork CLI/ODBC configuration keyword is set, CLI checks the current process ID for each function call for all applications that are connecting to the database for which the keyword is enabled. You can set this attribute so that CLI does not check for forked processes for that application.

The SQL_ATTR_PROCESSCTL attribute is an IBM defined extension.

SQL_ATTR_RESET_CONNECTION

Description

A 32-bit unsigned integer value that specifies whether the ODBC Driver Manager notifies the ODBC drivers that a connection has been placed in the connection pool on Windows operating systems. If the SQL_ATTR_ODBC_VERSION environment attribute is set to SQL_OV_ODBC3_80, the ODBC Driver Manager sets this attribute before placing a connection in the connection pool so that the driver can reset the other connection attributes to their default values.

Values

The only possible value is:

- SQL_RESET_CONNECTION_YES (default): The ODBC Driver Manager notifies the ODBC drivers that a connection has been placed in the connection pool.

Note: You should use SQL_ATTR_RESET_CONNECTION only for communication between the ODBC Driver Manager and an ODBC driver. You should not set this attribute from an application because all connection attributes will be reset to their default value. For example, any connection attribute values that you set by using the SQLSetConnectAttr () function will be reset to CLI default values and your application could behave unexpectedly.

SQL_ATTR_SYNC_POINT

This attribute was deprecated in DB2 database version 8.

This attribute is not supported when accessing the Informix database servers.

SQL_ATTR_TRACE

Description

A pointer to a null-terminated character string that is used to turn on the CLI/ODBC trace facility.

Values

The string must include the CLI keywords **TRACE** and **TRACEPATHNAME**. For example:

```
"TRACE=1; TRACEPATHNAME=<dir>;"
```

Usage notes

This attribute is not supported when accessing the Informix database servers.

SQL_ATTR_TRACENOHEADER

Description

A 32-bit integer value that specifies whether header information is included in the CLI trace file.

Values

The possible values are:

- **0** - Header information is included in the CLI trace file.
- **1** - No header information is included in the CLI trace file.

You can use the SQL_ATTR_TRACENOHEADER attribute with an SQL_NULL_HANDLE or with a valid environment handle.

SQL_ATTR_USE_2BYTES_OCTET_LENGTH

This attribute is deprecated in DB2 database version 8.

This attribute is not supported when accessing the Informix database servers.

SQL_ATTR_USE_LIGHT_OUTPUT_SQLDA

Setting this attribute is equivalent to setting the connection attribute SQL_ATTR_DESCRIBE_OUTPUT_LEVEL to 0. SQL_ATTR_USE_LIGHT_OUTPUT_SQLDA is deprecated and applications should now use the connection attribute SQL_ATTR_DESCRIBE_OUTPUT_LEVEL.

SQL_ATTR_USER_REGISTRY_NAME

Description

This attribute is used only when authenticating a user on a server that is using an identity mapping service.

Values

The SQL_ATTR_USER_REGISTRY_NAME attribute is set to a user defined string that names an identity mapping registry. The format of the name varies depending on the identity mapping service. By providing this attribute you tell the server that the user name that is provided can be found in this registry.

After setting this attribute, the value is used on subsequent attempts to establish a normal connection, establish a trusted connection, or switch the user ID on a trusted connection.

Usage notes

This attribute is not supported when accessing the Informix database servers.

SQL_CONNECTTYPE
This *Attribute* is replaced with SQL_ATTR_CONNECTTYPE.

SQL_MAXCONN
This *Attribute* is replaced with SQL_ATTR_MAXCONN.

SQL_SYNC_POINT
This *Attribute* is replaced with SQL_ATTR_SYNC_POINT.

This attribute is not supported when accessing the Informix database servers.

Connection attributes (CLI) list

The following table indicates when each of the CLI connection attributes can be set. A "Yes" in the "After statements allocated" column means that the connection attribute can be set both before and after the statements are allocated.

Table 167. When connection attributes can be set

Attribute	Before connection	After connection	After statements allocated
SQL_ATTR_ACCESS_MODE	Yes	Yes	Yes [a]
SQL_ATTR_ALLOW_INTERLEAVED_GETDATA	Yes	Yes	Yes
SQL_ATTR_ANSI_APP	Yes	No	No
SQL_ATTR_APP_USES_LOB_LOCATOR	Yes	Yes	Yes[c]
SQL_ATTR_APPEND_FOR_FETCH_ONLY	Yes	Yes	No
SQL_ATTR_ASYNC_ENABLE	Yes	Yes	Yes[a]
SQL_ATTR_AUTO_IPD (read-only)	No	No	No
SQL_ATTR_AUTOCOMMIT	Yes	Yes	Yes [b]
SQL_ATTR_CLIENT_CODEPAGE	Yes	No	No
SQL_ATTR_COLUMNWISE_MRI	Yes	Yes	Yes [a]
SQL_ATTR_COMMITONEOF	Yes	Yes	No
SQL_ATTR_CONCURRENT_ACCESS_RESOLUTION	Yes	Yes	Yes [a]
SQL_ATTR_CONN_CONTEXT	Yes	No	No
SQL_ATTR_CONNECT_NODE	Yes	No	No
SQL_ATTR_CONNECTION_DEAD (read-only)	No	No	No
SQL_ATTR_CONNECTTYPE	Yes	No	No
SQL_ATTR_CURRENT_CATALOG (read-only)	No	No	No
SQL_ATTR_CURRENT_IMPLICIT_XMLPARSE_OPTION	Yes	Yes	Yes
SQL_ATTR_CURRENT_PACKAGE_PATH	Yes	Yes	Yes
SQL_ATTR_CURRENT_PACKAGE_SET	Yes	Yes [a]	No [*]
SQL_ATTR_CURRENT_SCHEMA	Yes	Yes	Yes
SQL_ATTR_DB2_APPLICATION_HANDLE (read-only)	No	No	No
SQL_ATTR_DB2_APPLICATION_ID (read-only)	No	No	No
SQL_ATTR_DB2_SQLERRP (read-only)	No	No	No
SQL_ATTR_DB2EXPLAIN	No	Yes	Yes
SQL_ATTR_DECFLOAT_ROUNDING_MODE	Yes	Yes	Yes
SQL_ATTR_DESCRIBE_CALL	Yes	Yes	Yes[a]
SQL_ATTR_DESCRIBE_OUTPUT_LEVEL	Yes	Yes	No
SQL_ATTR_ENLIST_IN_DTC	No	Yes	Yes
SQL_ATTR_EXTENDED_INDICATORS	No	Yes	Yes
SQL_ATTR_FET_BUF_SIZE	Yes	No	No
SQL_ATTR_FREE_LOCATORS_ON_FETCH	Yes	Yes	Yes

Table 167. When connection attributes can be set (continued)

Attribute	Before connection	After connection	After statements allocated
SQL_ATTR_FORCE_ROLLBACK	Yes	Yes	Yes
SQL_ATTR_GET_LATEST_MEMBER	No	Yes	Yes
SQL_ATTR_INFO_ACCTSTR	No	Yes	Yes
SQL_ATTR_INFO_APPLNAME	No	Yes	Yes
SQL_ATTR_INFO_PROGRAMID	No	Yes	Yes [a]
SQL_ATTR_INFO_PROGRAMNAME	Yes	No	No
SQL_ATTR_INFO_USERID	No	Yes	Yes
SQL_ATTR_INFO_WRKSTNNAME	No	Yes	Yes
SQL_ATTR_KEEP_DYNAMIC	No	Yes	Yes
SQL_ATTR_LOB_CACHE_SIZE	Yes	Yes	Yes[c]
SQL_ATTR_LOGIN_TIMEOUT	Yes	No	No
SQL_ATTR_LONGDATA_COMPAT	Yes	Yes	Yes
SQL_ATTR_MAX_LOB_BLOCK_SIZE	Yes	Yes	Yes[c]
SQL_ATTR_MAPCHAR	Yes	Yes	Yes
SQL_ATTR_NETWORK_STATISTICS	Yes	Yes	Yes
SQL_ATTR_OVERRIDE_CHARACTER_CODEPAGE	No	Yes	No
SQL_ATTR_OVERRIDE_CODEPAGE	No	Yes	No
SQL_ATTR_PARC_BATCH	Yes	Yes	Yes[*]
SQL_ATTR_PING_DB (read only)	No	No	No
SQL_ATTR_PING_NTIMES	Yes	Yes	Yes
SQL_ATTR_PING_REQUEST_PACKET_SIZE	Yes	Yes	Yes
SQL_ATTR_QUIET_MODE	Yes	Yes	Yes
SQL_ATTR_RECEIVE_TIMEOUT	Yes	Yes	Yes
SQL_ATTR_REOPT	No	Yes	Yes[c]
SQL_ATTR_REPORT_ISLONG_FOR_LONGTYPES_OLEDB	Yes	Yes	Yes
SQL_ATTR_REPORT_SEAMLESSFAILOVER_WARNING	Yes	Yes	Yes[*]
SQL_ATTR_REPORT_TIMESTAMP_TRUNC_AS_WARN	Yes	Yes	Yes
SQL_ATTR_RETRYONERROR	Yes	Yes	Yes
SQL_ATTR_SERVER_MSGTXT_MASK	Yes	Yes	Yes
SQL_ATTR_SERVER_MSGTXT_SP	Yes	Yes	Yes
SQL_ATTR_SESSION_TIME_ZONE	Yes	No	No
SQL_ATTR_SQLCODEMAP	Yes	No	No
SQL_ATTR_SQLCOLUMNS_SORT_BY_ORDINAL_OLEDB	Yes	Yes	Yes
SQL_ATTR_STMT_CONCENTRATOR	Yes	Yes	Yes
SQL_ATTR_STREAM_GETDATA	Yes	Yes	Yes[c]
SQL_ATTR_TRUSTED_CONTEXT_PASSWORD	No	Yes	Yes
SQL_ATTR_TRUSTED_CONTEXT_USERID	No	Yes	Yes
SQL_ATTR_TXN_ISOLATION	No	Yes [b]	Yes [a]
SQL_ATTR_USE_TRUSTED_CONTEXT	Yes	No	No
SQL_ATTR_USER_REGISTRY_NAME	Yes	No	No
SQL_ATTR_WCHARTYPE	Yes	Yes [b]	Yes [b]
SQL_ATTR_XML_DECLARATION	Yes	Yes	Yes [a]

[a] Will only affect subsequently allocated statements.

[b] Attribute can be set only if there are no open transactions on the connection.

[c] Attribute can be set only if there are no open cursors on the connection. The attribute will affect all statements.

[*] Setting this attribute after statements have been allocated will not result in an error, however, determining which packages are used by which statements is ambiguous and unexpected behavior might occur. It is not recommended that you set this attribute after statements have been allocated.

Attribute
ValuePtr contents

SQL_ATTR_ACCESS_MODE
A 32-bit integer value which can be either:

- SQL_MODE_READ_ONLY: the application is indicating that it will not be performing any updates on data from this point on. Therefore, a less restrictive isolation level and locking can be used on transactions: uncommitted read (SQL_TXN_READ_UNCOMMITTED). CLI does not ensure that requests to the database are *read-only*. If an update request is issued, CLI will process it using the transaction isolation level it has selected as a result of the SQL_MODE_READ_ONLY setting.
- **SQL_MODE_READ_WRITE (default)**: the application is indicating that it will be making updates on data from this point on. CLI will go back to using the default transaction isolation level for this connection.

There must not be any outstanding transactions on this connection.

SQL_ATTR_ALLOW_INTERLEAVED_GETDATA
Specifies whether the application can call `SQLGetData()` for previously accessed LOB columns and maintain the data offset position from the previous call to `SQLGetData()` when querying data servers that support Dynamic Data Format. This attribute has one of the following values:

- SQL_ALLOW_INTERLEAVED_GETDATA_OFF - This default setting does not allow applications to call `SQLGetData()` for previously accessed LOB columns.
- SQL_ALLOW_INTERLEAVED_GETDATA_ON - This keyword only affects connections to database servers that support Dynamic Data Format, also known as progressive streaming. Specify this option to allow applications to call `SQLGetData()` for previously accessed LOB columns and start reading LOB data from where the application stopped reading during the previous read.

Setting the "AllowInterleavedGetData CLI/ODBC configuration keyword" on page 327 is an alternative method of specifying this behavior at the connection level.

SQL_ATTR_ANSI_APP
A 32-bit unsigned integer that identifies an application as an ANSI or Unicode application. This attribute has either of the following values:

- **SQL_AA_TRUE (default)**: the application is an ANSI application. All character data is passed to and from the application in the native application (client) code page using the ANSI version of the CLI/ODBC functions.
- SQL_AA_FALSE: the application is a Unicode application. All character data is passed to and from the application in Unicode when the Unicode (W) versions of the CLI/ODBC functions are called.

SQL_ATTR_APP_USES_LOB_LOCATOR
A 32-bit unsigned integer that indicates if applications are using LOB locators. This attribute has either of the following values:

- **1 (default)**: Indicates that applications are using LOB locators.
- 0: For applications that do not use LOB locators and are querying data on a server that supports Dynamic Data Format, also known as progressive streaming, specify 0 to indicate that LOB locators are not used and allow the return of LOB data to be optimized.

This keyword is ignored for stored procedure result sets.

If the keyword is set to 0 and an application binds a LOB locator to a result set using `SQLBindCol()`, an Invalid conversion error will be returned by the `SQLFetch()` function.

Setting the "AppUsesLOBLocator CLI/ODBC configuration keyword" on page 329 is an alternative method of specifying this behavior.

SQL_ATTR_APPEND_FOR_FETCH_ONLY

By default, CLI appends the "FOR FETCH ONLY" clause to read SELECT statements when connected to DB2 for z/OS or IBM DB2 for IBM i (DB2 for i) databases.

This attribute allows an application to control at a connection level when CLI appends the "FOR FETCH ONLY" clause. For example, an application is binding the CLI packages using different bind BLOCKING options (for example, BLOCKING UNAMBIG) and wants to suppress the blocking in order to keep positioned on a given row.

To change the default CLI behavior, the keyword is set as follows:

- 0: CLI never appends the "FOR FETCH ONLY" clause to read SELECT statements regardless of the server type it is connecting to.
- 1: CLI always appends the "FOR FETCH ONLY" clause to read SELECT statements regardless of the server type it is connecting to.

The attribute should be set either after the connection is allocated or immediately after it is established and should be set once for the duration of the execution of the application. Application can query the attribute with `SQLGetConnectAttr()` after connection is established or after this attribute is set.

Setting the "AppendForFetchOnly CLI/ODBC configuration keyword" on page 329 is an alternative method of specifying this behavior.

SQL_ATTR_ASYNC_ENABLE

A 32-bit integer value that specifies whether a function called with a statement on the specified connection is executed asynchronously:

- **SQL_ASYNC_ENABLE_OFF (default)** = Off
- SQL_ASYNC_ENABLE_ON = On

Setting SQL_ASYNC_ENABLE_ON enables asynchronous execution for all statement handles allocated on this connection. An error is returned if asynchronous execution is turned on while there is an active statement on the connection.

This attribute can be set whether `SQLGetInfo()`, called with the *InfoType* SQL_ASYNC_MODE, returns SQL_AM_CONNECTION or SQL_AM_STATEMENT.

Once a function has been called asynchronously, only the original function, `SQLAllocHandle()`, `SQLCancel()`, `SQLGetDiagField()`, or `SQLGetDiagRec()` can be called on the statement or the connection associated with *StatementHandle*, until the original function returns a code other than SQL_STILL_EXECUTING. Any other function called on *StatementHandle* or the connection associated with *StatementHandle* returns SQL_ERROR with an SQLSTATE of HY010 (Function sequence error).

The following functions can be executed asynchronously: `SQLBulkOperations()`, `SQLColAttribute()`, `SQLColumnPrivileges()`, `SQLColumns()`, `SQLDescribeCol()`, `SQLDescribeParam()`, `SQLExecDirect()`,

SQLExecute(), SQLExtendedFetch(), SQLExtendedPrepare(), SQLFetch(), SQLFetchScroll(), SQLForeignKeys(), SQLGetData(), SQLGetLength(), SQLGetPosition(), SQLMoreResults(), SQLNumResultCols(), SQLParamData(), SQLPrepare(), SQLPrimaryKeys(), SQLProcedureColumns(), SQLProcedures(), SQLRowCount(), SQLSetPos(), SQLSpecialColumns(), SQLStatistics(), SQLTablePrivileges(), SQLTables().

Note: Unicode equivalent functions can also be called asynchronously.

SQL_ATTR_AUTO_IPD

A read-only 32-bit unsigned integer value that specifies whether automatic population of the IPD after a call to SQLPrepare() is supported:

- SQL_TRUE = Automatic population of the IPD after a call to SQLPrepare() is supported by the server.
- SQL_FALSE = Automatic population of the IPD after a call to SQLPrepare() is not supported by the server. Servers that do not support prepared statements will not be able to populate the IPD automatically.

If SQL_TRUE is returned for the SQL_ATTR_AUTO_IPD connection attribute, the statement attribute SQL_ATTR_ENABLE_AUTO_IPD can be set to turn automatic population of the IPD on or off. If SQL_ATTR_AUTO_IPD is SQL_FALSE, SQL_ATTR_ENABLE_AUTO_IPD cannot be set to SQL_TRUE.

The default value of SQL_ATTR_ENABLE_AUTO_IPD is equal to the value of SQL_ATTR_AUTO_IPD.

This connection attribute can be returned by SQLGetConnectAttr(), but cannot be set by SQLSetConnectAttr().

SQL_ATTR_AUTOCOMMIT

A 32-bit unsigned integer value that specifies whether to use auto-commit or manual commit mode:

- SQL_AUTOCOMMIT_OFF: the application must manually, explicitly commit or rollback transactions with SQLEndTran() calls.
- **SQL_AUTOCOMMIT_ON (default)**: CLI operates in auto-commit mode by default. Each statement is implicitly committed. Each statement that is not a query is committed immediately after it has been executed or rolled back if failure occurred. Each query is committed immediately after the associated cursor is closed.

  Note: If this is a coordinated distributed unit of work connection, then the default is **SQL_AUTOCOMMIT_OFF**

Since in many DB2 environments, the execution of the SQL statements and the commit might be flowed separately to the database server, autocommit can be expensive. It is recommended that the application developer take this into consideration when selecting the auto-commit mode.

Note: Changing from manual commit to auto-commit mode will commit any open transaction on the connection.

SQL_ATTR_CLIENT_CODEPAGE

A connection level attribute value that enables the user to specify connection level code page from the CLI application. Specifying this attribute will override any environment level default code page setting:

Example 1: Setting the code page to be used by this database connection

```
Unicode = 1208;
cliRC = SQLSetConnectAttr(hdbc,
                          SQL_ATTR_CLIENT_CODEPAGE,
                          (SQLPOINTER)&unicode,  SQL_NTS);
char *connStr = "DSN=EBCDICDB;";
cliRC = SQLDriverConnect (hdbc, (SQLHWND)NULL, connStr, SQL_NTS, NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
```

Example 2: Getting the current value of SQL_ATTR_CLIENT_CODEPAGE

```
cliRC = SQLConnect(hdbc,
                           (SQLCHAR *)"SAMPLE",
                            SQL_NTS,
                           (SQLCHAR *)"USER1",
                           SQL_NTS,
                           (SQLCHAR *)"PASSWD1",
                           SQL_NTS);

cliRC = SQLGetConnectAttr(hdbc,
                          SQL_ATTR_CLIENT_CODEPAGE,
                          &codePage, 0, NULL);
```

SQL_ATTR_CLIENT_LOB_BUFFERING

Specifies whether LOB locators or the underlying LOB data is returned in a result set for LOB columns that are not bound. By default, locators are returned. If an application usually fetches unbound LOBs and then must retrieve the underlying LOB data, the application's performance can be improved by retrieving the LOB data from the outset; this reduces the number of synchronous waits and network flows. The possible values for this attribute are:

- SQL_CLIENTLOB_USE_LOCATORS (default) - LOB locators are returned
- SQL_CLIENTLOB_BUFFER_UNBOUND_LOBS - actual LOB data is returned

SQL_ATTR_CLIENT_TIME_ZONE

A null-terminated character string in the format ±hh:mm, containing the Time Zone information. Specifying this attribute will override the default Operating System Time Zone value of Client host.

SQL_ATTR_COLUMNWISE_MRI

A 32-bit unsigned integer that enables CLI applications connected to DB2 for z/OS servers to convert array input chaining into column-wise array input for INSERT operations. This attribute is available starting in Version 9.7 Fix Pack 5. The possible values are as follows:

- **SQL_COLUMNWISE_MRI_OFF (default)**: CLI does not convert chaining data to column-wise array input.
- SQL_COLUMNWISE_MRI_ON: CLI converts array input chaining to column-wise array input. The Multi-Row Insert (MRI) feature in DB2 for z/OS expects data to be in column-wise array form. If your application uses array input chaining, this conversion helps you optimize your application performance because data is sent in a compact array form each time you call SQLExecute (). For more information about array input chaining, see SQL_ATTR_CHAINING_BEGIN.

For outside a DB2 for z/OS servers, CLI automatically converts chaining data to row-wise array input and setting this attribute has no effect.

The conversion is not performed in the following cases:

- Bind parameters with a LOB data type such as SQL_CLOB, SQL_BLOB, SQL_LONGVARBINARY, SQL_LONGVARGRAPHIC, SQL_DBCLOB, or SQL_XML.

- Bind input data-at-execute parameters by setting their value to SQL_DATA_AT_EXEC to pass data to INSERT operations by calling the SQLPutData() and SQLParamData() functions.
- Space to store all the application data in the internal buffers is not available.

SQL_ATTR_COMMITONEOF

A 32-bit integer value that specifies whether or not to issue an implicit COMMIT immediately after reading the an entire result set and receiving an EOF. This connection attribute is available starting in Version 9.7 Fix Pack 5. The possible values for this attribute are:

- **SQL_ COMMITONEOF_OFF (default)**: A COMMIT is not implicitly issued after reading the entire result set. You have to explicitly call the SQLFreeStmt() function to close the cursor and release resources.
- SQL_ COMMITONEOF_ON: An implicit COMMIT is issued after reading the entire result set.

Note: Usage of this attribute does not replace the required call to the `SQLFreeStmt()` function.

SQL_ATTR_CONCURRENT_ACCESS_RESOLUTION

A 32-bit integer value that specifies the concurrent access resolution to use at the statement level. This setting overrides the default behavior specified for cursor stability (CS) scans.

- 0 = No setting. The client does not supply a prepare option.
- 1 = Use currently committed semantics. CLI flows "currently committed" on every prepare, which means that the database manager can use the currently committed version of the data for applicable scans when the data is in the process of being updated or deleted. Rows in the process of being inserted can be skipped. This setting applies when the isolation level in effect is Cursor Stability or Read Stability (for Read Stability it skips uncommitted inserts only) and is ignored otherwise. Applicable scans include read-only scans that can be part of a read-only statement as well as a non read-only statement. The settings for the registry variables **DB2_EVALUNCOMMITTED**, **DB2_SKIPDELETED**, and **DB2_SKIPINSERTED** do not apply to scans using currently committed. However, the settings for these registry variables still apply to scans that do not use currently committed.
- 2 = Wait for outcome. CLI flows "wait for outcome" on every prepare, which means that Cursor Stability and higher scans wait for the commit or rollback when encountering data in the process of being updated or deleted. Rows in the process of being inserted are not skipped. The settings for the registry variables DB2_EVALUNCOMMITTED, DB2_SKIPDELETED, and DB2_SKIPINSERTED no longer apply.
- 3 = Skip locked data. CLI flows "skip locked data" on every prepare, which means that currently committed semantics are used and rows in the process of being inserted are skipped. This option is not supported on DB2 Database for Linux, UNIX, and Windows. If specified, this setting is ignored.

For DB2 Database for Linux, UNIX, and Windows, use this attribute to override the default behavior for currently committed that is defined by the **cur_commit** configuration parameter. For DB2 for z/OS, use this attribute to enable currently committed behavior. There is no equivalent database configuration parameter available on DB2 for z/OS for specifying this behavior.

Setting the "ConcurrentAccessResolution CLI/ODBC configuration keyword" on page 341 is an alternative method of specifying this behavior.

SQL_ATTR_CONN_CONTEXT
Indicates which context the connection should use. An SQLPOINTER to either:

- a valid context (allocated by the `sqleBeginCtx()` DB2 API) to set the context
- a NULL pointer to reset the context

This attribute can only be used when the application is using the DB2 context APIs to manage multi-threaded applications. By default, CLI manages contexts by allocating one context per connection handle, and ensuring that any executing thread is attached to the correct context.

For more information about contexts, refer to the `sqleBeginCtx()` API.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_CONNECT_NODE
A 32-bit integer that specifies the target logical partition of a DB2 Enterprise Server Edition database partition server that you want to connect to. The possible values for this attribute are:

- an integer between 0 and 999
- SQL_CONN_CATALOG_NODE

If this variable is not set, the target logical node defaults to the logical node which is defined with port 0 on the machine.

This attribute is not supported when accessing IDS data servers.

There is also a corresponding keyword, the "ConnectNode CLI/ODBC configuration keyword" on page 342.

SQL_ATTR_CONNECTION_DEAD
A read only 32-bit integer value that indicates whether or not the connection is still active. CLI will return one of the following values:

- SQL_CD_FALSE - the connection is still active.
- SQL_CD_TRUE - an error has already happened and caused the connection to the server to be terminated. The application should still perform a disconnect to clean up any CLI resources.

This attribute is used mainly by the Microsoft ODBC Driver Manager 3.5x before pooling the connection.

SQL_ATTR_CONNECTION_TIMEOUT
This connection attribute is defined by ODBC, but is not supported by CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

SQL_ATTR_CONNECTTYPE
A 32-bit integer value that specifies whether this application is to operate in a coordinated or uncoordinated distributed environment. The possible values are as follows:

- **SQL_CONCURRENT_TRANS (default)**: The application can have concurrent multiple connections to any one database or to multiple databases. Each connection has its own commit scope. No effort is made to enforce coordination of transactions. If an application issues a commit

using the environment handle on SQLEndTran() and not all of the connections commit successfully, the application is responsible for recovery.

- SQL_COORDINATED_TRANS: The application wishes to have commit and rollbacks coordinated among multiple database connections. This option setting corresponds to the specification of the Type 2 CONNECT in embedded SQL. In contrast to the SQL_CONCURRENT_TRANS setting, the application is permitted only one open connection per database.

  Note: This connection type results in the default for SQL_ATTR_AUTOCOMMIT connection option to be SQL_AUTOCOMMIT_OFF.

If changing this attribute from the default then it must be set before any connections have been established on the environment handle.

It is recommended that the application set this attribute as an environment attribute with a call to SQLSetEnvAttr(), if necessary, as soon as the environment handle has been allocated. However, since ODBC applications cannot access SQLSetEnvAttr(), they must set this attribute using SQLSetConnectAttr() after each connection handle is allocated, but before any connections have been established.

All connections on an environment handle must have the same SQL_ATTR_CONNECTTYPE setting. An environment cannot have a mixture of concurrent and coordinated connections. The type of the first connection will determine the type of all subsequent connections. SQLSetEnvAttr() will return an error if an application attempts to change the connection type while there is an active connection.

The default connect type can also be set using the "ConnectType CLI/ODBC configuration keyword" on page 344.

Note: This is an IBM defined extension.

SQL_ATTR_CURRENT_CATALOG
A null-terminated character string containing the name of the catalog used by the data source. The catalog name is typically the same as the database name.

This connection attribute can be returned by SQLGetConnectAttr(), but cannot be set by SQLSetConnectAttr(). Any attempt to set this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

SQL_ATTR_CURRENT_IMPLICIT_XMLPARSE_OPTION
A null-terminated character string that is the string constant used to set the CURRENT IMPLICIT XMLPARSE OPTION special register. Setting this attribute causes the SET CURRENT IMPLICIT XMLPARSE OPTION SQL statement to be issued. If this attribute is set before a connection has been established, the SET CURRENT IMPLICIT XMLPARSE OPTION SQL statement will be issued when the connection is made.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_CURRENT_PACKAGE_PATH
A null-terminated character string of package qualifiers that the DB2 database server uses to try to resolve the package when multiple packages

have been configured. Setting this attribute causes the "SET CURRENT PACKAGE PATH = *schema1, schema2, ...*" statement to be issued after every connection to the database server.

This attribute is best suited for use with ODBC static processing applications, rather than CLI applications.

This attribute is not supported when accessing IDS data servers.

Note: This is an IBM defined extension.

SQL_ATTR_CURRENT_PACKAGE_SET

A null-terminated character string that indicates the schema name (collection identifier) that is used to select the package for subsequent SQL statements. Setting this attribute causes the SET CURRENT PACKAGESET SQL statement to be issued. If this attribute is set before a connection, the SET CURRENT PACKAGESET SQL statement will be issued at connection time.

CLI/ODBC applications issue dynamic SQL statements. Using this connection attribute, you can control the privileges used to run these statements:

- Choose a schema to use when running SQL statements from CLI/ODBC applications.
- Ensure the objects in the schema have the required privileges and then rebind accordingly. This typically means binding the CLI packages (sqllib/bnd/db2cli.lst) using the COLLECTION <collid> option. Refer to the BIND command for further details.
- Set the CURRENTPACKAGESET option to this schema.

The SQL statements from the CLI/ODBC applications will now run under the specified schema and use the privileges defined there.

Setting the “CurrentPackageSet CLI/ODBC configuration keyword” on page 347 is an alternative method of specifying the schema name.

The following package set names are reserved: NULLID, NULLIDR1, NULLIDRA.

SQL_ATTR_REOPT and SQL_ATTR_CURRENT_PACKAGE_SET are mutually exclusive, therefore, if one is set, the other is not allowed.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_CURRENT_SCHEMA

A null-terminated character string containing the name of the schema to be used by CLI for the `SQLColumns()` call if the *szSchemaName* pointer is set to null.

To reset this option, specify this option with a zero length string or a null pointer for the *ValuePtr* argument.

This option is useful when the application developer has coded a generic call to `SQLColumns()` that does not restrict the result set by schema name, but needs to constrain the result set at isolated places in the code.

This option can be set at any time and will be effective on the next `SQLColumns()` call where the *szSchemaName* pointer is null.

Note: This is an IBM defined extension.

SQL_ATTR_DB2_APPLICATION_HANDLE
A user-defined character string that returns the application handle of the connection. If the string is not large enough to contain the complete application handle, it will be truncated.

This connection attribute can be returned by `SQLGetConnectAttr()`, but cannot be set by `SQLSetConnectAttr()`.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_DB2_APPLICATION_ID
A user-defined character string that returns the application identifier of the connection. If the string is not large enough to contain the complete application identifier, it will be truncated.

This connection attribute can be returned by `SQLGetConnectAttr()`, but cannot be set by `SQLSetConnectAttr()`.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_DB2_SQLERRP
An sqlpointer to a null-terminated string containing the *sqlerrp* field of the sqlca.

Begins with a three-letter identifier indicating the product, followed by five alphanumeric characters indicating the version, release, and modification level of the product. The characters A-Z indicate a modification level higher than 9. A indicates modification level 10, B indicates modification level 11, and so on. For example, SQL0907C means DB2 Version 9 Release 7 Modification level 12.

If SQLCODE indicates an error condition, then this field identifies the module that returned the error.

This field is also used when a successful connection is completed.

Note: This is an IBM defined extension.

SQL_ATTR_DB2ESTIMATE
This attribute has been deprecated in DB2 UDB Version 8.

SQL_ATTR_DB2EXPLAIN
A 32-bit integer that specifies whether Explain snapshot, Explain mode information, or both should be generated by the server. Permitted values are:

- SQL_DB2EXPLAIN_OFF: Both the Explain Snapshot and the Explain table option facilities are disabled (a SET CURRENT EXPLAIN SNAPSHOT=NO and a SET CURRENT EXPLAIN MODE=NO are sent to the server).
- SQL_DB2EXPLAIN_SNAPSHOT_ON: The Explain Snapshot facility is enabled, and the Explain table option facility is disabled (a SET CURRENT EXPLAIN SNAPSHOT=YES and a SET CURRENT EXPLAIN MODE=NO are sent to the server).
- SQL_DB2EXPLAIN_MODE_ON: The Explain Snapshot facility is disabled, and the Explain table option facility is enabled (a SET CURRENT EXPLAIN SNAPSHOT=NO and a SET CURRENT EXPLAIN MODE=YES are sent to the server).

- SQL_DB2EXPLAIN_SNAPSHOT_MODE_ON: Both the Explain Snapshot and the Explain table option facilities are enabled (a SET CURRENT EXPLAIN SNAPSHOT=YES and a SET CURRENT EXPLAIN MODE=YES are sent to the server).

Before the explain information can be generated, the explain tables must be created.

This statement is not under transaction control and is not affected by a ROLLBACK. The new SQL_ATTR_DB2EXPLAIN setting is effective on the next statement preparation for this connection.

The current authorization ID must have INSERT privilege for the Explain tables.

The default value can also be set using the "DB2Explain CLI/ODBC configuration keyword" on page 350.

This attribute is not supported when accessing IDS data servers.

Note: This is an IBM defined extension.

SQL_ATTR_DECFLOAT_ROUNDING_MODE

The decimal float rounding mode determines what type of rounding will be used if a value is put into a DECFLOAT variable or column but the value has more digits than are allowed in the DECFLOAT data type. This can occur when inserting, updating, selecting, converting from another type, or as the result of a mathematical operation.

The value of SQL_ATTR_DECFLOAT_ROUNDING_MODE determines the decimal float rounding mode that will be used for new connections unless another mode is specified by a connection attribute for that connection. For any given connection both CLI and DB2 will use the same decimal float rounding mode for all action initiated as part of that connection.

When your applications are connecting to a DB2 Database for Linux, UNIX, and Windows Version 9.5 server, you must set the decimal float rounding mode on the database client to the same mode that is set on the server. If you set the decimal float rounding mode on the client to a value that is different from the decimal float rounding mode that is set on the database server, the database server will return SQL0713N on connection.

The settings correspond to these decimal float rounding modes:

- 0 = Half even (default)
- 1 = Half up
- 2 = Down
- 3 = Ceiling
- 4 = Floor

The different modes are:

Half even (default)
In this mode CLI and DB2 use the number that will fit in the target variable and that is closest to the original value. If two numbers are equally close, they use the one that is even. This mode produces the smallest rounding errors over large amounts of data.

Half up
In this mode CLI and DB2 use the number that will fit in the target

variable and that is closest to the original value. If two numbers are equally close, they use the one that is greater than the original value.

Down In this mode CLI and DB2 use the number that will fit in the target variable and that is closest to the original value and for which the absolute value is not greater than the absolute value of the original value. You can also think of this as rounding toward zero or as using ceiling for negative values and using floor for positive values.

Ceiling
In this mode CLI and DB2 use the smallest number that will fit in the target variable and that is greater than or equal to the original value.

Floor In this mode CLI and DB2 use the largest number that will fit in the target variable and that is less than or equal to the original value.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_DESCRIBE_CALL
A 32-bit integer value that indicates when stored procedure arguments are described. By default, CLI does not request input parameter describe information when it prepares a CALL statement. If an application has correctly bound parameters to a statement, then this describe information is unnecessary and not requesting it improves performance. The option values are:

- 1 = SQL_DESCRIBE_CALL_BEFORE.
- -1 = SQL_DESCRIBE_CALL_DEFAULT.

Setting this attribute can be done using the “DescribeCall CLI/ODBC configuration keyword” on page 356. Refer to the keyword for usage information and descriptions of the available options.

Note: This is an IBM defined extension.

SQL_ATTR_DESCRIBE_OUTPUT_LEVEL
A null-terminated character string that controls the amount of information the CLI driver requests on a prepare or describe request. By default, when the server receives a describe request, it returns the information contained in level 2 of Table 168 on page 449 for the result set columns. An application, however, might not need all of this information or might need additional information. Setting the SQL_ATTR_DESCRIBE_OUTPUT_LEVEL attribute to a level that suits the needs of the client application might improve performance because the describe data transferred between the client and server is limited to the minimum amount that the application requires. If the SQL_ATTR_DESCRIBE_OUTPUT_LEVEL setting is set too low, it might impact the functionality of the application (depending on the application's requirements). The CLI functions to retrieve the describe information might not fail in this case, but the information returned might be incomplete. Supported settings for SQL_ATTR_DESCRIBE_OUTPUT_LEVEL are:

- 0 - no describe information is returned to the client application
- 1 - describe information categorized in level 1 (see Table 168 on page 449) is returned to the client application

- 2 - (default) describe information categorized in level 2 (see Table 168) is returned to the client application
- 3 - describe information categorized in level 3 (see Table 168) is returned to the client application

The following table lists the fields that form the describe information that the server returns when it receives a prepare or describe request. These fields are grouped into levels, and the SQL_ATTR_DESCRIBE_OUTPUT_LEVEL attribute controls which levels of describe information the CLI driver requests.

Note:

1. Not all levels of describe information are supported by all DB2 servers. All levels of describe information are supported on the following DB2 servers: DB2 on Linux, UNIX, and Windows Version 8 and later, DB2 for z/OS Version 8 and later, and DB2 for i Version 5 Release 3 and later. All other DB2 servers support only the 2 or 0 setting for SQL_ATTR_DESCRIBE_OUTPUT_LEVEL.
2. The default behavior will allow CLI to promote the level to 3 if the application asks for describe information that was not initially retrieved using the default level 2. This might result in two network flows to the server. If an application uses this attribute to explicitly set a describe level, then no promotion will occur. Therefore, if the attribute is used to set the describe level to 2, then CLI will not promote to level 3 even if the application asks for extended information.

Table 168. Levels of describe information

Level 1	Level 2	Level 3
SQL_DESC_COUNT SQL_COLUMN_COUNT SQL_DESC_TYPE SQL_DESC_CONCISE_TYPE SQL_COLUMN_LENGTH SQL_DESC_OCTET_LENGTH SQL_DESC_LENGTH SQL_DESC_PRECISION SQL_COLUMN_PRECISION SQL_DESC_SCALE SQL_COLUMN_SCALE SQL_DESC_DISPLAY_SIZE SQL_DESC_NULLABLE SQL_COLUMN_NULLABLE SQL_DESC_UNSIGNED SQL_DESC_SEARCHABLE SQL_DESC_LITERAL_SUFFIX SQL_DESC_LITERAL_PREFIX SQL_DESC_CASE_SENSITIVE SQL_DESC_FIXED_PREC_SCALE	all fields of level 1 and: SQL_DESC_NAME SQL_DESC_LABEL SQL_COLUMN_NAME SQL_DESC_UNNAMED SQL_DESC_TYPE_NAME SQL_DESC_DISTINCT_TYPE SQL_DESC_REFERENCE_TYPE SQL_DESC_STRUCTURED_TYPE SQL_DESC_USER_TYPE SQL_DESC_LOCAL_TYPE_NAME SQL_DESC_USER_DEFINED_ TYPE_CODE	all fields of levels 1 and 2 and: SQL_DESC_BASE_COLUMN_NAME SQL_DESC_UPDATABLE SQL_DESC_AUTO_UNIQUE_VALUE SQL_DESC_SCHEMA_NAME SQL_DESC_CATALOG_NAME SQL_DESC_TABLE_NAME SQL_DESC_BASE_TABLE_NAME

Setting the "DescribeOutputLevel CLI/ODBC configuration keyword" on page 357 is an alternative method of specifying this behavior.

SQL_ATTR_ENLIST_IN_DTC

An SQLPOINTER which can be either of the following values:

- non-null transaction pointer: The application requests to CLI to change the state of the connection from non-distributed transaction state to distributed state. The connection is enlisted with the Distributed Transaction Coordinator (DTC).

- null: The application requests to CLI to change the state of the connection from distributed transaction state to a non-distributed transaction state.

This attribute is only used in a Microsoft Transaction Server (MTS) environment to enlist or un-enlist a connection with MTS.

Each time this attribute is used with a non-null transaction pointer, the previous transaction is assumed to be ended and a new transaction is initiated. The application must call the ITransaction member function Endtransaction before calling this API with a non-null pointer. Otherwise the previous transaction will be aborted. The application can enlist multiple connections with the same transaction pointer.

Note: This connection attribute is specified by MTS automatically for each transaction and is not coded by the user application.
It is imperative for CLI/ODBC applications that there will be no concurrent SQL statements executing on 2 different connections into the same database that are enlisted in the same transaction.

SQL_ATTR_EXTENDED_INDICATORS

A 32-bit integer that allows users to use the extended indicator feature from the supported server. If the user attempts to set this attribute against the data server which does not support extended indicators, an appropriate error is returned to the CLI application. This attribute can take the following value:

- SQL_EXTENDED_INDICATOR_ENABLE: Enables users to specify values to signify SQL_UNASSIGNED and SQL_DEFAULT_PARAM on the SQLBindParameter() / SQLExtendedBind() methods.
- SQL_EXTENDED_INDICATOR_NOT_SET (default): This feature is disabled by default. The user gets an InvalidArgument value error, CLI0124E, if the SQL_UNASSIGNED and SQL_DEFAULT_PARAM are used before enabling this feature using SQL_ATTR_EXTENDED_INDICATORS.
- Extended indicators support DB2 for Linux, UNIX, and Windows and for DB2 10 for z/OS data servers starts in DB2 Version 9.7 Fix Pack 2. Extended indicators support DB2 for IBM i 7.1 data servers starts in DB2 Version 9.7 Fix Pack 5.

SQL_ATTR_FET_BUF_SIZE

A connection level attribute to allow applications to set the default query block size to an optimum value in range of 64K-256K. This attribute should be set before a connection is made. CLI will also provide a db2cli.ini level keyword, FET_BUF_SIZE, which can be set in `db2cli.ini` file and connection string.

An equivalent db2dsdriver.cfg keyword, FetchBufferSize is also available, which can be set in the `db2dsdriver.cfg` file.

SQL_ATTR_FREE_LOCATORS_ON_FETCH

A boolean attribute that specifies if LOB locators are freed when `SQLFetch()` is executed, rather than when a COMMIT is issued. Setting this attribute to 1 (true) frees the locators that are used internally when applications fetch LOB data without binding the LOB columns with `SQLBindCol()` (or equivalent descriptor APIs). Locators that are explicitly returned to the application must still be freed by the application. This attribute value can be used to avoid scenarios where an application receives SQLCODE = -429 (no more locators). The default for this attribute is 0 (false).

Note: This is an IBM defined extension.

SQL_ATTR_FORCE_ROLLBACK
A 32-bit unsigned integer value that allows calls to the SQLEndTran() function in a data-at-execution flow for connections to DB2 for z/OS and OS/390 servers.

To call the SQLEndTran() function specifying SQL_ROLLBACK as *CompletionType* in your applications during a data-at-execution flow, the StreamPutData configuration keyword must be set to 1, and the SQL_ATTR_FORCE_ROLLBACK connection attribute must also be set.

The CLI0150E error message is returned for connections to data servers that are not DB2 for z/OS and OS/390 servers.

Note: This is an IBM defined extension.

SQL_ATTR_GET_LATEST_MEMBER
A connection level attribute, which is available starting in Version 9.7, Fix Pack 3, enables CLI applications to retrieve the latest member (being) used for the given connection.

CLI applications can retrieve the currently connected or last connected member on a connection, by calling the SQLGetConnectAttr() function.

The CLI0126E error message is returned if you try to use this attribute before establishing a database connection.

SQL_ATTR_INFO_ACCTSTR
A pointer to a null-terminated character string used to identify the client accounting string sent to the data server when using DB2 Connect or DB2 Database for Linux, UNIX, and Windows.

Please note:

- When the value is being set, some servers might not handle the entire length provided and might truncate the value.
- DB2 for z/OS and OS/390 servers support up to a length of 200 characters.
- To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.)

Note: This is an IBM defined extension.

SQL_ATTR_INFO_APPLNAME
A pointer to a null-terminated character string used to identify the client application name sent to the data server when using DB2 Connect or DB2 database products for Linux, UNIX and Windows.

Please note:

- When the value is being set, some servers might not handle the entire length provided and might truncate the value.
- DB2 for z/OS and OS/390 servers support up to a length of 32 characters.
- To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

Note: This is an IBM defined extension.

SQL_ATTR_INFO_PROGRAMID
A user-defined character string, with a maximum length of 80 bytes, that associates an application with a connection. Once this attribute is set, DB2 UDB for z/OS Version 8 and later associates this identifier with any statements inserted into the dynamic SQL statement cache.

This attribute is only supported for CLI applications accessing DB2 UDB for z/OS Version 8 and later or IBM Informix Dynamic Servers (IDS).

Note: This is an IBM defined extension.

SQL_ATTR_INFO_PROGRAMNAME
A null-terminated user-defined character string, up to 20 bytes in length, used to specify the name of the application running on the client.

When this attribute is set before the connection to the server is established, the value specified overrides the actual client application name and will be the value that is displayed in the appl_name monitor element. When connecting to a DB2 for z/OS server, the first 12 characters of this setting are used as the CORRELATION IDENTIFIER of the associated DB2 for z/OS thread.

Note: This is an IBM defined extension.

SQL_ATTR_INFO_USERID
A pointer to a null-terminated character string used to identify the client user ID sent to the data server when using DB2 Connect or DB2 database products for Linux, UNIX and Windows.

Please note:

- When the value is being set, some servers might not handle the entire length provided and might truncate the value.
- DB2 for z/OS and OS/390 servers support up to a length of 16 characters.
- This user-id is not to be confused with the authentication user-id. This user-id is for identification purposes only and is not used for any authorization.
- To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

Note: This is an IBM defined extension.

SQL_ATTR_INFO_WRKSTNNAME
A pointer to a null-terminated character string used to identify the client workstation name sent to the data server when using DB2 Connect or DB2 database products for Linux, UNIX and Windows.

Please note:

- When the value is being set, some servers might not handle the entire length provided and might truncate the value.
- DB2 for z/OS and OS/390 servers support up to a length of 18 characters.
- To ensure that the data is converted correctly when transmitted to a host system, use only the characters A to Z, 0 to 9, and the underscore (_) or period (.).

Note: This is an IBM defined extension.

SQL_ATTR_KEEP_DYNAMIC
A 32-bit unsigned integer value which specifies whether the KEEPDYNAMIC option has been enabled. If enabled, the server will keep dynamically prepared statements in a prepared state across transaction boundaries.

- 0 - KEEPDYNAMIC functionality is not available; CLI packages were bound with the KEEPDYNAMIC NO option
- 1 - KEEPDYNAMIC functionality is available; CLI packages were bound with the KEEPDYNAMIC YES option

It is recommended that when this attribute is set, the SQL_ATTR_CURRENT_PACKAGE_SET attribute also be set.

This attribute is not supported when accessing IDS data servers.

Note: This is an IBM defined extension.

SQL_ATTR_LOB_CACHE_SIZE
A 32-bit unsigned integer that specifies maximum cache size (in bytes) for LOBs. By default, LOBs are not cached.

See the "LOBCacheSize CLI/ODBC configuration keyword" on page 367 for further usage information.

SQL_ATTR_LOGIN_TIMEOUT
A 32-bit integer value corresponding to the number of seconds to wait for a reply when trying to establish a connection to a server before terminating the attempt and generating a communication timeout. Specify a positive integer, up to 32 767. The default setting of 0 will allow the client to wait indefinitely.

Setting a connection timeout value can also be done using the "ConnectTimeout CLI/ODBC configuration keyword" on page 343. Refer to the keyword for usage information.

SQL_ATTR_LONGDATA_COMPAT
A 32-bit integer value indicating whether the character, double byte character and binary large object data types should be reported as SQL_LONGVARCHAR, SQL_LONGVARGRAPHIC or SQL_LONGBINARY, enabling existing applications to access large object data types seamlessly. The option values are:

- **SQL_LD_COMPAT_NO (default)**: The large object data types are reported as IBM defined types (SQL_BLOB, SQL_CLOB, SQL_DBCLOB).
- SQL_LD_COMPAT_YES: The IBM large object data types (SQL_BLOB, SQL_CLOB and SQL_DBCLOB) are mapped to SQL_LONGVARBINARY, SQL_LONGVARCHAR and SQL_LONGVARGRAPHIC; `SQLGetTypeInfo()` returns one entry each for SQL_LONGVARBINARY SQL_LONGVARCHAR, and SQL_LONGVARGRAPHIC.

Note: This is an IBM defined extension.

SQL_ATTR_MAPCHAR
A 32-bit integer value used to specify the default SQL type associated with SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR. The option values are:

- **SQL_MAPCHAR_DEFAULT (default)**: return the default SQL type representation

- SQL_MAPCHAR_WCHAR: return SQL_CHAR as SQL_WCHAR, SQL_VARCHAR as SQL_WVARCHAR, and SQL_LONGVARCHAR as SQL_WLONGVARCHAR

Only the following CLI functions are affected by setting this attribute:

- SQLColumns()
- SQLColAttribute()
- SQLDescribeCol()
- SQLDescribeParam()
- SQLGetDescField()
- SQLGetDescRec()
- SQLProcedureColumns()

Setting the default SQL type associated with SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR can also be done using the "MapCharToWChar CLI/ODBC configuration keyword" on page 370.

Note: This is an IBM defined extension.

SQL_ATTR_MAXCONN
This attribute has been deprecated in DB2 UDB Version 8.

SQL_ATTR_MAX_LOB_BLOCK_SIZE
A 32-bit unsigned integer that indicates the maximum size of LOB or XML data block. Specify a positive integer, up to 2 147 483 647. The default setting of 0 indicates that there is no limit to the data block size for LOB or XML data.

During data retrieval, the server will include all of the information for the current row in its reply to the client even if the maximum block size has been reached.

If both MaxLOBBlockSize and the db2set registry variable DB2_MAX_LOB_BLOCK_SIZE are specified, the value for MaxLOBBlockSize will be used.

Setting the "MaxLOBBlockSize CLI/ODBC configuration keyword" on page 377 is an alternative method of specifying this behavior.

SQL_ATTR_METADATA_ID
This connection attribute is defined by ODBC, but is not supported by CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

SQL_ATTR_NETWORK_STATISTICS
Starting in Version 9.7, Fix Pack 3, this 32-bit integer connection attribute controls whether CLI collects network statistics for a connection. An application can retrieve the network statistics for a connection by calling the SQLGetDiagField() function and specifying SQL_DIAG_NETWORK_STATISTICS for the *DiagIdentifier* argument.

The permitted values are as follows:

SQL_NETWORK_STATISTICS_OFF (default)
Disables network statistics collection for a connection.

SQL_NETWORK_STATISTICS_ON
Enables network statistics collection for a connection.

SQL_NETWORK_STATISTICS_ON_SKIP_NOSERVER
In addition to enabling network statistics collection for a

connection, network flows are omitted that are known to have no server time reported, for example explicit COMMIT and ROLLBACK statements.

Requests that have no server time reported can affect the usefulness of returned information, if calculations are made that subtract the server time from the network time. The SQL_NETWORK_STATISTICS_ON_SKIP_NOSERVER option excludes these requests from the values reported. Only explicit, unchained requests are excluded; autocommit and chained COMMIT statements are not skipped.

Starting in Version 9.7 Fix Pack 5, CLI collects statistics for server time reported on COMMIT and ROLLBACK. The DB2 server must be at a level that supports reporting server time for COMMIT and ROLLBACK.

SQL_ATTR_ODBC_CURSORS
This connection attribute is defined by ODBC, but is not supported by CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

SQL_ATTR_OVERRIDE_CODEPAGE
This connection attribute, which is available starting in Version 9.7 Fix Pack 5, enables CLI applications to fetch or insert data of CHARACTER or GRAPHIC data type without code page conversions. The possible values are as follows:

- SQL_OVERRIDE_CODEPAGE_ON: CLI does not perform code page conversions for binding of character or graphic data.
- **SQL_OVERRIDE_CODEPAGE_OFF** (default): CLI performs code page conversions for binding of character or graphic data.

If you set this attribute to SQL_OVERRIDE_CODEPAGE_ON, you must ensure that data is in the correct code page.

If you set SQL_ATTR_OVERRIDE_CODEPAGE to SQL_OVERRIDE_CODEPAGE_ON after setting SQL_ATTR_OVERRIDE_CHARACTER_CODEPAGE, CLI returns the CLI0126E error message.

SQL_ATTR_OVERRIDE_CHARACTER_CODEPAGE
This connection attribute, which is available starting in Version 9.7 Fix Pack 3, enables CLI applications to specify the database code page. The code page does not have to be available at the client end.

If you specify the same code page as the database code page, applications can fetch or insert data of CHARACTER data type without any code page conversions.

Setting this attribute after allocating statement handles results in error CLI0126E (invalid operation). Setting the attribute to a value that is not supported by the database results in error CLI0210E (inconsistent code page value error).

If you set SQL_ATTR_OVERRIDE_CHARACTER_CODEPAGE after setting SQL_ATTR_OVERRIDE_CODEPAGE to SQL_OVERRIDE_CODEPAGE_ON, CLI returns the CLI0126E error message.

During a bind-out operation, ensure that the CLI applications allocate buffers large enough to hold the retrieved data during bind-out operations. If there is insufficient space, error CLI0002W is returned.

Restriction: This attribute is supported only for DB2 for z/OS data servers. If you attempt to set the value of this attribute for other data servers, error CLI0150E (driver not capable) is returned.

SQL_ATTR_PACKET_SIZE
This connection attribute is defined by ODBC, but is not supported by CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

SQL_ATTR_PARC_BATCH
For applications that use array input to achieve bulk inserts, deletes, or updates, this 32-bit unsigned integer connection attribute indicates whether the application receives the number of rows in a table that were affected by the each parameter set or the cumulative number of rows that were affected for the entire parameter set. This connection attribute is available starting in DB2 Version 9.7 Fix Pack 5 for non-atomic operations. The possible values are as follows:

- SQL_PARC_BATCH_ENABLE: CLI returns the number of rows in a table that were affected by the each parameter set.
- **SQL_PARC_BATCH_DISABLE (default)**: CLI returns the total number of rows that were affected for the entire parameter set.

If you set this connection attribute to SQL_PARCH_BATCH_ENABLE, you must indicate an array as the *RowCountPtr* parameter in the SQLRowCount () function and you must set SQL_ATTR_PARAMOPT_ATOMIC to SQL_ATOMIC_NO. If SQL_ATTR_PARAMOPT_ATOMIC is set to SQL_ATOMIC_YES, the CLI0150E error message is returned when you call the SQLExecute () function.

SQL_ATTR_PING_DB
A 32-bit integer which is used with `SQLGetConnectAttr()` to get the ping time in microseconds.

If a connection has previously been established and has been dropped by the database, a value of 0 is reported. If the connection has been closed by the application, then an SQLSTATE of 08003 is reported. This connection attribute can be returned by `SQLGetConnectAttr()`, but cannot be set by `SQLSetConnectAttr()`. Any attempt to set this attribute will result in an SQLSTATE of 7HYC00 (Driver not capable)

Note: This is an IBM defined extension.

SQL_ATTR_PING_NTIMES
A 32-bit integer that is used with `SQLGetConnectAttr()` that sets the number of ping iterations that CLI performs when the application uses SQL_ATTR_PING_DB. If you set SQL_ATTR_PING_NTIMES to a value greater than 1, SQL_ATTR_PING_DB returns the average time that CLI took to ping the database for the set of iterations.

This attribute has a valid range from 1 to 32767 (inclusive). `SQLGetConnectAttr()` checks the value and returns the appropriate error code when the value is outside this range.

SQL_ATTR_PING_REQUEST_PACKET_SIZE
A 32-bit integer that is used with `SQLGetConnectAttr()` that sets the size of the ping packet that CLI uses when the application uses SQL_ATTR_PING_DB.

This attribute has a valid range from 1 to 32767 (inclusive).`SQLGetConnectAttr()` checks the value and returns the appropriate error code when the value is outside this range.

SQL_ATTR_QUIET_MODE
A 32-bit platform specific window handle.

If the application has never made a call to `SQLSetConnectAttr()` with this option, then CLI would return a null parent window handle on `SQLGetConnectAttr()` for this option and use a null parent window handle to display dialogue boxes. For example, if the end user has asked for (via an entry in the CLI initialization file) optimizer information to be displayed, CLI would display the dialogue box containing this information using a null window handle. (For some platforms, this means the dialogue box would be centered in the middle of the screen.)

If *ValuePtr* is set to null , then CLI does not display any dialogue boxes. In cases where end user asked for the optimizer estimates to be displayed, CLI would not display these estimates because the application explicitly wants to suppress all such dialogue boxes.

If *ValuePtr* is not null, then it should be the parent window handle of the application. CLI uses this handle to display dialogue boxes. (For some platforms, this means the dialogue box would be centered with respect to the active window of the application.)

Note: This connection option cannot be used to suppress the `SQLDriverConnect()` dialogue box (which can be suppressed by setting the *fDriverCompletion* argument to SQL_DRIVER_NOPROMPT).

SQL_ATTR_RECEIVE_TIMEOUT

A 32-bit integer value that is the number of seconds a client waits for a reply from a server on an established connection before terminating the attempt and generating a communication timeout error. The default value of 0 indicates the client waits indefinitely for a reply. The receive timeout has no effect during connection establishment; it is only supported for TCP/IP, and is ignored for any other protocol. Supported values are integers from 0 to 32767.

Note: This is an IBM defined extension.

SQL_ATTR_REOPT

A 32-bit integer value that enables query optimization for SQL statements that contain special registers or parameter markers. Optimization occurs by using the values available at query execution time for special registers or parameter markers, instead of the default estimates that are chosen by the compiler. The valid values of the attribute are:

- **2 = SQL_REOPT_NONE (default):** No query optimization occurs at query execution time. The default estimates chosen by the compiler are used for the special registers or parameter markers. The default NULLID package set is used to execute dynamic SQL statements.

- 3 = SQL_REOPT_ONCE: Query optimization occurs once at query execution time, when the query is executed for the first time. The NULLIDR1 package set, which is bound with the REOPT ONCE bind option, is used.
- 4 = SQL_REOPT_ALWAYS: Query optimization or reoptimization occurs at query execution time every time the query is executed. The NULLIDRA package set, which is bound with the REOPT ALWAYS bind option, is used.

The NULLIDR1 and NULLIDRA are reserved package set names, and when used, REOPT ONCE and REOPT ALWAYS are implied. These package sets have to be explicitly created with these commands:

```
db2 bind db2clipk.bnd collection NULLIDR1
db2 bind db2clipk.bnd collection NULLIDRA
```

SQL_ATTR_REOPT and SQL_ATTR_CURRENT_PACKAGE_SET are mutually exclusive, therefore, if one is set, the other is not allowed.

This attribute is not supported when accessing IDS data servers.

Note: This is an IBM defined extension.

SQL_ATTR_REPORT_ISLONG_FOR_LONGTYPES_OLEDB
A 32-bit integer value. The OLE DB client cursor engine and the OLE DB .NET Data Provider CommandBuilder object generate UPDATE and DELETE statements based on column information provided by the IBM DB2 OLE DB Provider. If the generated statement contains a LONG type in the WHERE clause, the statement will fail because LONG types cannot be used in a search with an equality operator. The possible values are as follows:

- **0 (default)**: LONG types (LONG VARCHAR, LONG VARCHAR FOR BIT DATA, LONG VARGRAPHIC and LONG VARGRAPHIC FOR BIT DATA) do not have the DBCOLUMNFLAGS_ISLONG flag set, which might cause the columns to be used in the WHERE clause.
- 1: The IBM DB2 OLE DB Provider reports LONG types (LONG VARCHAR, LONG VARCHAR FOR BIT DATA, LONG VARGRAPHIC and LONG VARGRAPHIC FOR BIT DATA) with the DBCOLUMNFLAGS_ISLONG flag set. This will prevent the long columns from being used in the WHERE clause.

This attribute is supported by the following database servers:

- DB2 for z/OS
  - version 6 with PTF UQ93891
  - version 7 with PTF UQ93889
  - version 8 with PTF UQ93890
  - versions later than version 8, PTFs are not required
- DB2 Database for Linux, UNIX, and Windows
  - version 8.2 (equivalent to Version 8.1, FixPak 7) and later

This attribute is not supported when accessing IDS data servers.

Note: This is an IBM defined extension.

SQL_ATTR_REPORT_SEAMLESSFAILOVER_WARNING
A 32-bit unsigned integer value that specifies whether to return a warning message on execute requests if a seamless failover occurred during the

request. This connection attribute is available starting in DB2 Version 9.7 Fix Pack 5. The possible values are as follows:

- SQL_REPORT_SEAMLESSFAILOVER_WARNING_YES: If a seamless failover occurred during an execute request, a warning message is returned.
- **SQL_REPORT_SEAMLESSFAILOVER_WARNING_NO (default)**: If a seamless failover occurred during an execute request, a warning message is not returned.

SQL_ATTR_REPORT_TIMESTAMP_TRUNC_AS_WARN
A 32-bit unsigned integer value that specifies whether a datetime overflow results in an error (SQLSTATE 22008) or warning (SQLSTATE 01S07). The possible values are as follows:

- **0 (default)**: Datetime overflow results in an error (SQLSTATE 22008).
- 1: Datetime overflow results in a warning (SQLSTATE 01S07).

SQL_ATTR_RETRYONERROR
CLI attempts recover from non-fatal errors, such as incorrect binding of application parameters, by retrieving additional information about the failing SQL statement and then executing the statement again. The additional information retrieved includes input parameter information from the database catalog tables. If CLI is able to recover successfully from the error, by default, it does not report the error to the application. The CLI/ODBC configuration keyword ReportRetryErrorsAsWarnings allows you to set whether error recovery warnings are returned to the application or not.

Note: Once CLI has successfully completed the error recovery, the application may behave differently, because CLI uses the catalog information gathered during the recovery for subsequent executions of that particular SQL statement, rather than the information provided in the original SQLBindParameter() function calls. If you do not want this behavior, set RetryOnError to 0, forcing CLI not to attempt recovery. You should, however, modify the application to correctly bind statement parameters..

SQL_ATTR_SERVER_MSGTXT_MASK
A 32-bit integer value used to indicate when CLI should request the error message from the server. This attribute is used in conjunction with the SQL_ATTR_SERVER_MSGTXT_SP attribute. The attribute can be set to:

- **SQL_ATTR_SERVER_MSGTXT_MASK_LOCAL_FIRST (default)**: CLI will check the local message files first to see if the message can be retrieved. If no matching SQLCODE is found, then CLI will request the information from the server.
- SQL_ATTR_SERVER_MSGTXT_MASK_WARNINGS: CLI always requests the message information from the server for warnings but error messages are retrieved from the local message files.
- SQL_ATTR_SERVER_MSGTXT_MASK_ERRORS: CLI always requests the message information from the server for errors but warning messages are retrieved from the local message files.
- SQL_ATTR_SERVER_MSGTXT_MASK_ALL: CLI always requests the message information from the server for both error and warning messages.

Setting the “ServerMsgMask CLI/ODBC configuration keyword” on page 398 is an alternative method of specifying this behavior.

Note: This is an IBM defined extension.

SQL_ATTR_SERVER_MSGTXT_SP
A pointer to a character string used to identify a stored procedure that is used for generating an error message based on an SQLCA. This can be useful when retrieving error information from a server such as DB2 for z/OS. The attribute can be set to:

- SYSIBM.SQLCAMESSAGE: The default procedure called to retrieve the message text from DB2 for z/OS servers. If you do not set this attribute, this procedure is called.
- DSNACCMG: The default procedure called to retrieve the message text from DB2 for z/OS Version 7 servers. The SYSIBM.SQLCAMESSAGE procedure is called to retrieve the message text from DB2 for z/OS Version 8 or later. DSNACCMG has been deprecated in DB2 for z/OS Version 9 and might be removed in a future release.
- Any user-created stored procedure.

Applications using this attribute can also set the SQL_ATTR_SERVER_MSGTXT_MASK attribute to indicate when CLI should call this procedure to retrieve the message information from the server. If the SQL_ATTR_SERVER_MSGTXT_MASK is not set, then the default is to check the local message files first (see SQL_ATTR_SERVER_MSGTXT_MASK_LOCAL_FIRST in SQL_ATTR_SERVER_MSGTXT_MASK).

Setting the "UseServerMsgSP CLI/ODBC configuration keyword" on page 424 is an alternative method of specifying this behavior.

Note: This is an IBM defined extension.

SQL_ATTR_SESSION_TIME_ZONE
A null-terminated character string in the format ±hh:mm, containing the server session time zone information. This is a set-only attribute. The supported time zone values range from -12:59 through +14:00.

SQL_ATTR_SQLCODEMAP
Specifies whether SQLCODE mapping should be set to default or turned off. The possible values are as follows:

- **MAP (default)**: SQLCODE mapping is set.
- NOMAP: SQLCODEMapping is turned off.

SQL_ATTR_SQLCOLUMNS_SORT_BY_ORDINAL_OLEDB
A 32–bit integer value. The Microsoft OLE DB specification requires that IDBSchemaRowset::GetRowset(DBSCHEMA_COLUMNS) returns the row set sorted by the columns TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME. The IBM DB2 OLE DB Provider conforms to the specification, however, applications that use the Microsoft ODBC Bridge provider (MSDASQL) have been typically coded to get the row set sorted by ORDINAL_POSITION. The possible values are as follows:

- **0 (default)**: The IBM DB2 OLE DB Provider returns a row set sorted by the columns TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME.
- 1: The IBM DB2 OLE DB Provider returns a row set sorted by ORDINAL_POSITION.

This attribute is supported by the following database servers:

- DB2 for z/OS
  - version 6 with PTF UQ93891
  - version 7 with PTF UQ93889
  - version 8 with PTF UQ93890
  - versions later than version 8, PTFs are not required
- DB2 Database for Linux, UNIX, and Windows
  - version 8.2 (equivalent to Version 8.1, FixPak 7) and later

This attribute is not supported when accessing IDS data servers.

Note: This is an IBM defined extension.

SQL_ATTR_STMT_CONCENTRATOR
Specifies whether dynamic statements that contain literal values use the statement cache.

- SQL_STMT_CONCENTRATOR_OFF - The statement concentrator behavior is disabled.
- SQL_STMT_CONCENTRATOR_WITH_LITERALS - The statement concentrator with literal behavior is enabled for situations that are supported by the server. For example, the statement concentrator is not enabled if the statement has parameter markers, named parameter markers, or a mix of literals, parameter markers, and named parameter markers.

Setting the "StmtConcentrator CLI/ODBC configuration keyword" on page 403 is an alternative method of specifying this behavior.

SQL_ATTR_STREAM_GETDATA
A 32-bit unsigned integer that indicates if the data output stream for the `SQLGetData()` function will be optimized. The values are:

- **0 (default)**: CLI buffers all the data on the client.
- 1: For applications that do not need to buffer data and are querying data on a server that supports Dynamic Data Format, also known as progressive streaming, specify 1 to indicate that data buffering is not required. The CLI client will optimize the data output stream.

This keyword is ignored if Dynamic Data Format is not supported by the server.

If StreamGetData is set to 1 and CLI cannot determine the number of bytes still available to return in the output buffer, `SQLGetData()` returns SQL_NO_TOTAL (-4) as the length when truncation occurs. Otherwise, `SQLGetData()` returns the number of bytes still available.

Setting the "StreamGetData CLI/ODBC configuration keyword" on page 404 is an alternative method of specifying this behavior.

SQL_ATTR_SYNC_POINT
This attribute has been deprecated in DB2 UDB Version 8.

SQL_ATTR_TRACE
This connection attribute can be set by an application for the ODBC Driver Manager. Any attempt to set this connection attribute for the CLI Driver will result in an SQLSTATE of HYC00 (Driver not capable).

Instead of using this connection attribute, the CLI trace facility can be set using the "Trace CLI/ODBC configuration keyword" on page 408. Alternatively, the environment attribute SQL_ATTR_TRACE can be used to

configure tracing features. Note that the environment attribute does not use the same syntax as the ODBC Driver Manager's connection attribute.

SQL_ATTR_TRACEFILE
This connection attribute is defined by ODBC, but is not supported by CLI. Any attempt to set or get this attribute will result in an SQLSTATE of HYC00 (Driver not capable).

Instead of using this attribute, the CLI trace file name is set using the "TraceFileName CLI/ODBC configuration keyword" on page 414.

SQL_ATTR_TRANSLATE_LIB
This connection attribute is defined by ODBC, but is not supported by CLI. Any attempt to set or get this attribute on other platforms will result in an SQLSTATE of HYC00 (Driver not capable).

SQL_ATTR_TRANSLATE_OPTION
This connection attribute is defined by ODBC, but is not supported by CLI. Any attempt to set or get this attribute on other platforms will result in an SQLSTATE of HYC00 (Driver not capable).

SQL_ATTR_TRUSTED_CONTEXT_PASSWORD
A user defined string containing a password. Use this attribute if the database server requires a password when switching users on a trusted connection. Set this attribute after setting the attribute SQL_ATTR_TRUSTED_CONTEXT_USERID and before executing any SQL statements that access the database server. If SQL_ATTR_TRUSTED_CONTEXT_USERID is not set before setting this attribute, an error (CLI0198E) is returned.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_TRUSTED_CONTEXT_USERID

A user defined string containing a user ID. Use this on existing trusted connections to switch users. Do not use it when creating a trusted connection.

After setting this attribute the user switch will occur the next time that you execute an SQL statement that accesses the database server. (SQLSetConnectAttr does not access the database server.) If the user switch is successful the user ID in this attribute becomes the new user of the connection. If the user switch fails the call that initiated the switch will return an error indicating the reason for the failure.

The user ID must be a valid authorization ID on the database server unless you are using an identity server, in which case you can use any user name recognized by the identity server. (If you are using an identity server see also SQL_ATTR_USER_REGISTRY_NAME.)

If you set this attribute while the connection handle is not yet connected to a database or if the connection is not a trusted connection then an error (CLI0197E) is returned.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_TXN_ISOLATION
A 32-bit bitmask that sets the transaction isolation level for the current connection referenced by *ConnectionHandle*. The valid values for *ValuePtr* can be determined at runtime by calling `SQLGetInfo()` with *fInfoType* set to SQL_TXN_ISOLATION_OPTIONS. The following values are accepted by CLI, but each server might only support a subset of these isolation levels:

- SQL_TXN_READ_UNCOMMITTED - Dirty reads, non-repeatable reads, and phantom reads are possible.
- **SQL_TXN_READ_COMMITTED (default)** - Dirty reads are not possible. Non-repeatable reads and phantom reads are possible.
- SQL_TXN_REPEATABLE_READ - Dirty reads and reads that cannot be repeated are not possible. Phantoms are possible.
- SQL_TXN_SERIALIZABLE - Transactions can be serialized. Dirty reads, non-repeatable reads, and phantoms are not possible.
- SQL_TXN_NOCOMMIT - Any changes are effectively committed at the end of a successful operation; no explicit commit or rollback is allowed. This is analogous to autocommit. This is not an SQL92 isolation level, but an IBM defined extension, supported only by DB2 UDB for AS/400.

In IBM terminology,
- SQL_TXN_READ_UNCOMMITTED is Uncommitted Read;
- SQL_TXN_READ_COMMITTED is Cursor Stability;
- SQL_TXN_REPEATABLE_READ is Read Stability;
- SQL_TXN_SERIALIZABLE is Repeatable Read.

This option cannot be specified while there is an open cursor on any statement handle, or an outstanding transaction for this connection; otherwise, SQL_ERROR is returned on the function call (SQLSTATE **S1**011).

This attribute (or corresponding keyword) is only applicable if the default isolation level is used. If the application has specifically set the isolation level then this attribute will have no effect.

Note: There is an IBM extension that permits the setting of transaction isolation levels on a per statement handle basis. See the SQL_ATTR_STMTTXN_ISOLATION statement attribute.

SQL_ATTR_USE_TRUSTED_CONTEXT

When connecting to a DB2 database server that supports trusted contexts, set this attribute if you want the connection you are creating to be a trusted connection. If this attribute is set to SQL_TRUE and the database server determines that the connection can be trusted then the connection is a trusted connection. If this attribute is not set, if it is set to SQL_FALSE, if the database server does not support trusted contexts, or if the database server determines that the connection cannot be trusted then a regular connection is created instead and a warning (SQLSTATE 01679) is returned. This value can only be specified before the connection is established either for the first time or following a call to the `SQLDisconnect()` function.

SQL_ATTR_USER_REGISTRY_NAME

This attribute is only used when authenticating a user on a server that is using an identity mapping service. It is set to a user defined string that names an identity mapping registry. The format of the registry name varies depending on the identity mapping service used. By providing this attribute you tell the server that the user name provided can be found in this registry.

After setting this attribute the value will be used on subsequent attempts to establish a normal connection, establish a trusted connection, or switch the user id on a trusted connection.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_WCHARTYPE

A 32-bit integer that specifies, in a double-byte environment, which wchar_t (SQLDBCHAR) character format you want to use in your application. This option provides you the flexibility to choose between having your wchar_t data in multi-byte format or in wide-character format. There two possible values for this option:

- SQL_WCHARTYPE_CONVERT: character codes are converted between the graphic SQL data in the database and the application variable. This allows your application to fully exploit the ANSI C mechanisms for dealing with wide character strings (for example, L-literals, 'wc' string functions) without having to explicitly convert the data to multi-byte format before communicating with the database. The disadvantage is that the implicit conversions might have an impact on the runtime performance of your application, and might increase memory requirements. If you want WCHARTYPE CONVERT behavior then define the C preprocessor macro SQL_WCHART_CONVERT at compile time. This ensures that certain definitions in the DB2 header files use the data type wchar_t instead of sqldbchar.
- **SQL_WCHARTYPE_NOCONVERT (default)**: no implicit character code conversion occurs between the application and the database. Data in the application variable is sent to and received from the database as unaltered DBCS characters. This allows the application to have improved performance, but the disadvantage is that the application must either refrain from using wide-character data in wchar_t (SQLDBCHAR) application variables, or it must explicitly call the wcstombs() and mbstowcs() ANSI C functions to convert the data to and from multi-byte format when exchanging data with the database.

Note: This is an IBM defined extension.

SQL_ATTR_XML_DECLARATION

A 32-bit unsigned integer that specifies which elements of an XML declaration are added to XML data when it is implicitly serialized. This attribute does not affect the result of the XMLSERIALIZE function. Set this attribute to the sum of each component required:

- 0: No declarations or byte order marks (BOMs) are added to the output buffer.
- 1: A byte order mark (BOM) in the appropriate endianness is prepended to the output buffer if the target encoding is UTF-16 or UTF-32. (Although a UTF-8 BOM exists, DB2 does not generate it, even if the target encoding is UTF-8.)
- 2: A minimal XML declaration is generated, containing only the XML version.
- 4: An encoding attribute that identifies the target encoding is added to any generated XML declaration. Therefore, this setting only has effect when the setting of 2 is also included when computing the value of this attribute.

Attempts to set any other value using `SQLSetConnectAttr()` or `SQLSetConnectOption()` will result in a CLI0191E (SQLSTATE HY024) error, and the value will remain unchanged.

The default setting is 7, which indicates that a BOM and an XML declaration containing the XML version and encoding attribute are generated during implicit serialization.

This setting affects any statement handles allocated after the value is changed. Existing statement handles retain their original values.

This attribute is not supported when accessing IDS data servers.

Statement attributes (CLI) list

The currently defined attributes and the version of CLI or ODBC in which they were introduced are shown in this section; it is expected that more will be defined to take advantage of different data sources.

SQL_ATTR_ALLOW_INTERLEAVED_GETDATA
Specifies whether the application can call `SQLGetData()` for previously accessed LOB columns and maintain the data offset position from the previous call to `SQLGetData()` when querying data servers that support Dynamic Data Format. This attribute has one of the following values:

- SQL_ALLOW_INTERLEAVED_GETDATA_OFF - This default setting does not allow applications to call `SQLGetData()` for previously accessed LOB columns.
- SQL_ALLOW_INTERLEAVED_GETDATA_ON - This keyword only affects connections to database servers that support Dynamic Data Format, also known as progressive streaming. Specify this option to allow applications to call `SQLGetData()` for previously accessed LOB columns and start reading LOB data from where the application stopped reading during the previous read.

Setting the "AllowInterleavedGetData CLI/ODBC configuration keyword" on page 327 is an alternative method of specifying this behavior at the connection level.

SQL_ATTR_ALLOW_INTERLEAVED_GETDATA connection attribute is not supported with an IDS data server.

SQL_ATTR_APP_PARAM_DESC
The handle to the APD for subsequent calls to `SQLExecute()` and `SQLExecDirect()` on the statement handle. The initial value of this attribute is the descriptor implicitly allocated when the statement was initially allocated. If this attribute is set to SQL_NULL_DESC, an explicitly allocated APD handle that was previously associated with the statement handle is dissociated from it, and the statement handle reverts to the implicitly allocated APD handle.

This attribute cannot be set to a descriptor handle that was implicitly allocated for another statement or to another descriptor handle that was implicitly set on the same statement; implicitly allocated descriptor handles cannot be associated with more than one statement or descriptor handle.

This attribute cannot be set at the connection level.

SQL_ATTR_APP_ROW_DESC
The handle to the ARD for subsequent fetches on the statement handle. The initial value of this attribute is the descriptor implicitly allocated when the statement was initially allocated. If this attribute is set to SQL_NULL_DESC, an explicitly allocated ARD handle that was previously associated with the statement handle is dissociated from it, and the statement handle reverts to the implicitly allocated ARD handle.

This attribute cannot be set to a descriptor handle that was implicitly allocated for another statement or to another descriptor handle that was

implicitly set on the same statement; implicitly allocated descriptor handles cannot be associated with more than one statement or descriptor handle.

This attribute cannot be set at the connection level.

SQL_ATTR_APP_USES_LOB_LOCATOR
A 32-bit unsigned integer that indicates if applications are using LOB locators. This attribute has either of the following values:

- **1 (default)**: Indicates that applications are using LOB locators.
- 0: For applications that do not use LOB locators and are querying data on a server that supports Dynamic Data Format, also known as progressive streaming, specify 0 to indicate that LOB locators are not used and allow the return of LOB data to be optimized.

This keyword is ignored for stored procedure result sets.

If the keyword is set to 0 and an application binds a LOB locator to a result set that uses `SQLBindCol()`, an Invalid conversion error is returned by the `SQLFetch()` function.

Setting the "AppUsesLOBLocator CLI/ODBC configuration keyword" on page 329 is an alternative method of specifying this behavior.

SQL_ATTR_ASYNC_ENABLE
A 32-bit integer value that specifies whether a function called with the specified statement is executed asynchronously:

- **SQL_ASYNC_ENABLE_OFF** = Off (the default)
- SQL_ASYNC_ENABLE_ON = On

After a function has been called asynchronously, only the original function, `SQLAllocHandle()`, `SQLCancel()`, `SQLSetStmtAttr()`, `SQLGetDiagField()`, `SQLGetDiagRec()`, or `SQLGetFunctions()` can be called on the statement handle, until the original function returns a code other than SQL_STILL_EXECUTING. Any other function called on any other statement handle under the same connection returns SQL_ERROR with an SQLSTATE of HY010 (Function sequence error).

Because CLI supports statement level asynchronous-execution, the statement attribute SQL_ATTR_ASYNC_ENABLE can be set. Its initial value is the same as the value of the connection level attribute with the same name at the time the statement handle was allocated.

The following functions can be executed asynchronously:
`SQLBulkOperations()`, `SQLColAttribute()`, `SQLColumnPrivileges()`, `SQLColumns()`, `SQLDescribeCol()`, `SQLDescribeParam()`, `SQLExecDirect()`, `SQLExecute()`, `SQLExtendedFetch()`, `SQLExtendedPrepare()`, `SQLFetch()`, `SQLFetchScroll()`, `SQLForeignKeys()`, `SQLGetData()`, `SQLGetLength()`, `SQLGetPosition()`, `SQLMoreResults()`, `SQLNumResultCols()`, `SQLParamData()`, `SQLPrepare()`, `SQLPrimaryKeys()`, `SQLProcedureColumns()`, `SQLProcedures()`, `SQLRowCount()`, `SQLSetPos()`, `SQLSpecialColumns()`, `SQLStatistics()`, `SQLTablePrivileges()`, `SQLTables()`.

Note: Any Unicode equivalent functions can also be called asynchronously. Starting from Version 9.7, Fix Pack 4, the SQL_ATTR_ASYNC_ENABLE attribute can be used with SQL_ATTR_USE_LOAD_API.

SQL_ATTR_BLOCK_FOR_NROWS
A 32-bit integer that specifies the required block size, in rows, to be returned by the server when fetching a result set. For large read-only result sets consisting of one or more data blocks, a large block size can improve

performance by reducing the number of synchronous server block requests made by the client. The default value is 0 which means the default block size is returned by the server.

SQL_ATTR_BLOCK_LOBS
A Boolean attribute that specifies if blocking of result sets returning LOB data types is enabled. By default, this attribute is set to 0 (false), however, when set to 1 (true) and when accessing a server that supports blocking of result sets returning LOB data types, all of the LOB data associated with rows that fit completely within a single query block are returned in a single fetch request.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_CALL_RETURN
A read-only attribute to be retrieved after executing a stored procedure. The value returned from this attribute is -1 if the stored procedure failed to execute (for example, if the library containing the stored procedure executable cannot be found). If the stored procedure executed successfully but has a negative return code (for example, if data truncation occurred when inserting data into a table), then SQL_ATTR_CALL_RETURN returns the value that was set in the sqlerrd (1) field of the SQLCA when the stored procedure was executed.

SQL_ATTR_CHAINING_BEGIN
A 32-bit integer which specifies that DB2 chains together `SQLExecute()` requests for a single prepared statement before sending the requests to the server; this feature is referred to as CLI array input chaining. All `SQLExecute()` requests associated with a prepared statement are not sent to the server until either the SQL_ATTR_CHAINING_END statement attribute is set, or the available buffer space is consumed by rows that have been chained. The size of this buffer is defined by the **aslheapsz** database manager configuration parameter for local client applications, or the **rqrioblk** database manager configuration parameter for client/server configurations.

This attribute can be used with the CLI/ODBC configuration keyword **ArrayInputChain** to effect array input without needing to specify the array size. See the documentation for **ArrayInputChain** for more information.

Note: The specific 32-bit integer value that is set with this attribute is not significant to CLI. Setting this attribute to any 32-bit integer value enables the CLI array input chaining feature.

SQL_ATTR_CHAINING_END
A 32-bit integer which specifies that the CLI array input chaining behavior enabled earlier, with the setting of the SQL_ATTR_CHAINING_BEGIN statement attribute, ends. Setting SQL_ATTR_CHAINING_END causes all chained `SQLExecute()` requests to be sent to the server. After this attribute is set, `SQLRowCount()` can be called to determine the total row count for all `SQLExecute()` statements that were chained between the SQL_ATTR_CHAINING_BEGIN and SQL_ATTR_CHAINING_END pair. Error diagnostic information for the chained statements becomes available after the SQL_ATTR_CHAINING_END attribute is set.

This attribute can be used with the CLI configuration keyword **ArrayInputChain** to affect array input without needing to specify the array size. See the documentation for **ArrayInputChain** for more information.

Note: The specific 32-bit integer value that is set with this attribute is not significant to CLI. Setting this attribute to any 32-bit integer value disables the CLI array input chaining feature that was enabled when SQL_ATTR_CHAINING_BEGIN was set.

SQL_ATTR_CLIENT_LOB_BUFFERING
Specifies whether LOB locators or the underlying LOB data is returned in a result set for LOB columns that are not bound. By default, locators are returned. If an application usually fetches unbound LOBs and then must retrieve the underlying LOB data, the application performance can be improved by retrieving the LOB data from the outset. This action reduces the number of synchronous waits and network flows. The possible values for this attribute are:

- SQL_CLIENTLOB_USE_LOCATORS (default) - LOB locators are returned
- SQL_CLIENTLOB_BUFFER_UNBOUND_LOBS - actual LOB data is returned

SQL_ATTR_CLOSE_BEHAVIOR
A 32-bit integer that specifies whether the DB2 server should attempt to release read locks acquired during a cursor's operation when the cursor is closed. It can be set to either:

- **SQL_CC_NO_RELEASE** - read locks are not released. This is the default.
- SQL_CC_RELEASE - read locks are released.

For cursors opened with isolation UR or CS, read locks are not held after a cursor moves off a row. For cursors opened with isolation RS or RR, SQL_ATTR_CLOSE_BEHAVIOR modifies some of those isolation levels, and an RR cursor might experience nonrepeatable reads or phantom reads.

If a cursor that is originally RR or RS is reopened after being closed with SQL_ATTR_CLOSE_BEHAVIOR then new read locks are acquired.

This attribute can also be set at the connection level, however when set at the connection level, it only affects cursor behavior for statement handles that are opened after this attribute is set.

See the `SQLCloseCursor()` function for more information.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_CLOSEOPEN
To reduce the time it takes to open and close cursors, DB2 automatically closes an open cursor if a second cursor is opened using the same handle. Network flow is therefore reduced when the close request is chained with the open request and the two statements are combined into one network request (instead of two requests).

- **0** = DB2 acts as a regular ODBC data source: Do not chain the close and open statements, return an error if there is an open cursor. This behavior is the default.
- 1 = Chain the close and open statements.

Previous CLI applications do not benefit from this default because they are designed to explicitly close the cursor. New applications, however, can take advantage of this behavior by not closing the cursors explicitly, but by allowing CLI to close the cursor on subsequent open requests.

SQL_ATTR_COLUMNWISE_MRI
A 32-bit unsigned integer that enables CLI applications connected to DB2

for z/OS servers to convert array input chaining into column-wise array input for INSERT operations. This attribute is available starting in Version 9.7 Fix Pack 5. The possible values are as follows:

- **SQL_COLUMNWISE_MRI_OFF (default)**: CLI does not convert chaining data to column-wise array input.
- SQL_COLUMNWISE_MRI_ON: CLI converts array input chaining to column-wise array input. The Multi-Row Insert (MRI) feature in DB2 for z/OS expects data to be in column-wise array form. If your application uses array input chaining, this conversion helps you optimize your application performance because data is sent in a compact array form each time you call SQLExecute (). For more information about array input chaining, see SQL_ATTR_CHAINING_BEGIN.

For outside a DB2 for z/OS servers, CLI automatically converts chaining data to row-wise array input and setting this attribute has no effect.

The conversion is not performed in the following cases:

- Bind parameters with a LOB data type such as SQL_CLOB, SQL_BLOB, SQL_LONGVARBINARY, SQL_LONGVARGRAPHIC, SQL_DBCLOB, or SQL_XML.
- Bind input data-at-execute parameters by setting their value to SQL_DATA_AT_EXEC to pass data to INSERT operations by calling the SQLPutData() and SQLParamData() functions.
- Space to store all the application data in the internal buffers is not available.

SQL_ATTR_CONCURRENCY

A 32-bit integer value that specifies the cursor concurrency:

- SQL_CONCUR_READ_ONLY = Cursor is read-only. No updates are allowed. Supported by forward-only, static and keyset cursors.
- SQL_CONCUR_LOCK = Cursor uses the lowest level of locking sufficient to ensure that the row can be updated. Supported by forward-only and keyset cursors.
- SQL_CONCUR_VALUES = Cursor uses optimistic concurrency control, comparing values.

The default value for SQL_ATTR_CONCURRENCY is SQL_CONCUR_READ_ONLY for static and forward-only cursors. The default for a keyset cursor is SQL_CONCUR_VALUES.

This attribute cannot be specified for an open cursor.

If the SQL_ATTR_CURSOR_TYPE *Attribute* is changed to a type that does not support the current value of SQL_ATTR_CONCURRENCY, the value of SQL_ATTR_CONCURRENCY is changed at execution time, and a warning issued when `SQLExecDirect()` or `SQLPrepare()` is called.

If a SELECT FOR UPDATE statement is executed while the value of SQL_ATTR_CONCURRENCY is set to SQL_CONCUR_READ_ONLY, an error is returned. If the value of SQL_ATTR_CONCURRENCY is changed to a value that is supported for some value of SQL_ATTR_CURSOR_TYPE, but not for the current value of SQL_ATTR_CURSOR_TYPE, the value of SQL_ATTR_CURSOR_TYPE is changed at execution time, and SQLSTATE 01S02 (Option value changed) is issued when `SQLExecDirect()` or `SQLPrepare()` is called.

If the value of SQL_ATTR_CONCURRENCY is set to SQL_CONCUR_LOCK, this value is promoted to SQL_CONCUR_VALUES when all the following conditions are met:

- SQL_ROWSET_SIZE OR SQL_ATTR_ROW_ARRAY_SIZE is greater than 1.
- The data source is a database on a DB2 Database for Linux, UNIX, and Windows server.
- The **PATCH2** configuration keyword is set to 73.

If the specified concurrency is not supported by the data source, then CLI substitutes a different concurrency and returns SQLSTATE 01S02 (Option value changed). The order of substitution depends on the cursor type:

- Forward-Only: SQL_CONCUR_LOCK is substituted for SQL_CONCUR_ROWVER and SQL_CONCUR_VALUES
- Static: only SQL_CONCUR_READ_ONLY is valid
- Keyset: SQL_CONCUR_VALUES is substituted for SQL_CONCUR_ROWVER

Note: The following value has also been defined by ODBC, but is not supported by CLI

- SQL_CONCUR_ROWVER = Cursor uses optimistic concurrency control.

SQL_ATTR_CURSOR_HOLD

A 32-bit integer which specifies whether the cursor associated with this *StatementHandle* is preserved in the same position as before the COMMIT operation, and whether the application can fetch without executing the statement again.

- **SQL_CURSOR_HOLD_ON** (this is the default)
- SQL_CURSOR_HOLD_OFF

The default value when an *StatementHandle* is first allocated is SQL_CURSOR_HOLD_ON.

This option cannot be specified while there is an open cursor on this *StatementHandle*.

The default cursor hold mode can also be set using the **CURSORHOLD** CLI/ODBC configuration keyword.

Note: This option is an IBM extension.

SQL_ATTR_CURSOR_SCROLLABLE

A 32-bit integer that specifies the level of support that the application requires. Setting this attribute affects subsequent calls to `SQLExecute()` and `SQLExecDirect()`. The supported values are:

- **SQL_NONSCROLLABLE** = Scrollable cursors are not required on the statement handle. If the application calls `SQLFetchScroll()` on this handle, the only valid value of *FetchOrientation()* is SQL_FETCH_NEXT. This value is the default.
- SQL_SCROLLABLE = Scrollable cursors are required on the statement handle. When calling `SQLFetchScroll()`, the application can specify any valid value of *FetchOrientation*, achieving cursor positioning in modes other than the sequential mode.

SQL_ATTR_CURSOR_SENSITIVITY

A 32-bit integer that specifies whether cursors on the statement handle make visible the changes made to a result set by another cursor. Setting this attribute affects subsequent calls to `SQLExecute()` and `SQLExecDirect()`. The supported values are:

- **SQL_UNSPECIFIED** = It is unspecified what the cursor type is and whether cursors on the statement handle make visible the changes made to a result set by another cursor. Cursors on the statement handle might make visible none, some or all such changes. This value is the default.
- SQL_INSENSITIVE = All cursors on the statement handle show the result set without reflecting any changes made to it by any other cursor. Insensitive cursors are read-only. This corresponds to a static cursor which has a concurrency that is read-only.
- SQL_SENSITIVE = All cursors on the statement handle make visible all changes made to a result by another cursor.

SQL_ATTR_CURSOR_TYPE

A 32-bit integer value that specifies the cursor type. The supported values are:

- **SQL_CURSOR_FORWARD_ONLY** = The cursor only scrolls forward. This is the default.
- SQL_CURSOR_STATIC = The data in the result set is static.
- SQL_CURSOR_KEYSET_DRIVEN = CLI supports a pure keyset cursor. The SQL_KEYSET_SIZE statement attribute is ignored. To limit the size of the keyset the application must limit the size of the result set by setting the SQL_ATTR_MAX_ROWS attribute to a value other than 0.
- SQL_CURSOR_DYNAMIC = A dynamic scrollable cursor detects all changes (inserts, deletes and updates) to the result set, and make insertions, deletions and updates to the result set. Dynamic cursors are only supported when accessing servers which are DB2 for z/OS Version 8.1 and later.

This option cannot be specified for an open cursor.

If the specified cursor type is not supported by the data source, CLI substitutes a different cursor type and returns SQLSTATE 01S02 (Option value changed). For a mixed or dynamic cursor, CLI substitutes, in order, a keyset-driven or static cursor.

SQL_ATTR_DB2_NOBINDOUT

A Boolean attribute that specifies when and where the client performs data conversion and related tasks during a fetch operation. The default value of this attribute is 0 (false) and should only be set to 1 (true) when connected to a federated database.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_DEFERRED_PREPARE

Specifies whether the PREPARE request is deferred until the corresponding execute request is issued.

- SQL_DEFERRED_PREPARE_OFF = Disable deferred prepare. The PREPARE request is executed the moment it is issued.
- **SQL_DEFERRED_PREPARE_ON** (default) = Enable deferred prepare. Defer the execution of the PREPARE request until the corresponding execute request is issued. The two requests are then combined into one command/reply flow (instead of two) to minimize network flow and to improve performance.

  If the target DB2 database or the DDCS gateway does not support deferred prepare, the client disables deferred prepare for that connection.

Note: When deferred prepare is enabled, the row and cost estimates normally returned in the SQLERRD(3) and SQLERRD(4) of the SQLCA of a

PREPARE statement might become zeros. This might be of concern to users who want to use these values to decide whether to continue the SQL statement.

The default deferred prepare mode can also be set using the **DEFERREDPREPARE** CLI/ODBC configuration keyword.

Note: This is an IBM defined extension.

SQL_ATTR_EARLYCLOSE
Specifies whether the temporary cursor on the server can be automatically closed, without closing the cursor on the client, when the last record is sent to the client.

- SQL_EARLYCLOSE_OFF = Do not close the temporary cursor on the server early.
- **SQL_EARLYCLOSE_ON** = Close the temporary cursor on the server early (default).

  This saves a network request by not issuing the statement to explicitly close the cursor because it knows that it has already been closed.

  Having this option on speeds up applications that use many small result sets.

  The EARLYCLOSE feature is not used if the cursor type is anything other than SQL_CURSOR_FORWARD_ONLY.

Note: This is an IBM defined extension.

SQL_ATTR_ENABLE_AUTO_IPD
A 32-bit integer value that specifies whether automatic population of the IPD is performed:

- SQL_TRUE = Turns on automatic population of the IPD after a call to `SQLPrepare()`.
- SQL_FALSE = Turns off automatic population of the IPD after a call to `SQLPrepare()`.

The default value of the statement attribute SQL_ATTR_ENABLE_AUTO_IPD is equal to the value of the connection attribute SQL_ATTR_AUTO_IPD.

If the connection attribute SQL_ATTR_ AUTO_IPD is SQL_FALSE, the statement attribute SQL_ATTR_ENABLE_AUTO_IPD cannot be set to SQL_TRUE.

SQL_ATTR_EXTENDED_INDICATORS
A 32-bit integer that eliminates the need to indicate the position in the SQL statement where the contents of the application variables are substituted when the statement is executed. This attribute has the following values:

- SQL_EXTENDED_INDICATOR_ENABLE = Enables users to specify values to signify SQL_UNASSIGNED and SQL_DEFAULT_PARAM on the SQLBindParameter / SQLExtendedBind methods.
- SQL_EXTENDED_INDICATOR_NOT_SET (default): The user gets an InvalidArgument value error if the SQL_UNASSIGNED and SQL_DEFAULT_PARAM are not enabled before an application tries to use them.
- Extended indicators support DB2 for Linux, UNIX, and Windows and for DB2 10 for z/OS data servers starts in DB2 Version 9.7 Fix Pack 2. Extended indicators support DB2 for IBM i 7.1 data servers starts in DB2 Version 9.7 Fix Pack 5.

SQL_ATTR_FETCH_BOOKMARK_PTR
A pointer that points to a binary bookmark value. When SQLFetchScroll() is called with *FetchOrientation* equal to SQL_FETCH_BOOKMARK, CLI picks up the bookmark value from this field. This field defaults to a null pointer.

SQL_ATTR_IMP_PARAM_DESC
The handle to the IPD. The value of this attribute is the descriptor allocated when the statement was initially allocated. The application cannot set this attribute.

This attribute can be retrieved by a call to SQLGetStmtAttr(), but not set by a call to SQLSetStmtAttr().

SQL_ATTR_IMP_ROW_DESC
The handle to the IRD. The value of this attribute is the descriptor allocated when the statement was initially allocated. The application cannot set this attribute.

This attribute can be retrieved by a call to SQLGetStmtAttr(), but not set by a call to SQLSetStmtAttr().

SQL_ATTR_INFO_PROGRAMID
A user-defined character string, with a maximum length of 80 bytes, that associates an application with a statement. Once this attribute is set, DB2 UDB for z/OS Version 8 and later associates this identifier with any statements inserted into the dynamic SQL statement cache.

This attribute is only supported for CLI applications accessing DB2 UDB for z/OS Version 8 and later or IBM Informix Dynamic Servers (IDS).

SQL_ATTR_INSERT_BUFFERING
This attribute enables buffering insert optimization of partitioned database environments. The possible values are:
SQL_ATTR_INSERT_BUFFERING_OFF (default),
SQL_ATTR_INSERT_BUFFERING_ON, and
SQL_ATTR_INSERT_BUFFERING_IGD (duplicates are ignored).

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_INTERLEAVED_PUTDATA
This attribute allows inserting LOB data with SQLParamData and SQLPutData in an interleaving fashion. For example:

```
// Set the attribute
SQLSetStmtAttr(hstmt,
               SQL_ATTR_INTERLEAVED_PUTDATA,
               TRUE,
               0);

//Bind the parameters with DATA_AT_EXEC indicator
blobInd = SQL_DATA_AT_EXEC;

cliRC = SQLBindParameter (hstmt,          /* statement handle      */
                          1,              /* parameter marker index */
                          SQL_PARAM_INPUT, /* it's input parameter  */
                          SQL_C_CHAR,      /* CLI variable is CHARACTER*/
                          SQL_CLOB,        /* table column is CLOB*/
                          10,              /* length of CLI variable */
                          0,              /* scale of decimal digits*/
                          &data1,          /* pointer to CLI variable*/
                          10,              /* buffer length          */
                          &blobInd);


cliRC = SQLBindParameter (hstmt,          /* statement handle      */
```

```
2,              /* parameter marker index */
SQL_PARAM_INPUT, /* it's input parameter  */
SQL_C_CHAR,      /* CLI variable is CHARACTER*/
SQL_CLOB,        /* table column is CLOB*/
10,              /* length of CLI variable */
0,              /* scale of decimal digits*/
&data2,          /* pointer to CLI variable*/
10,              /* buffer length          */
&blobInd);
```

```
SQLExecute (hstmt);
valuePtr = (SQLPOINTER) 2;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);
//update buffer data2
SQLPutData (hstmt, data2, strlen(data2));
valuePtr = (SQLPOINTER) 1;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);
//update buffer data1
SQLPutData (hstmt, data1, strlen(data2));
valuePtr = (SQLPOINTER) 2;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);
//update buffer data2
SQLPutData (hstmt, data2, strlen(data2));
valuePtr = (SQLPOINTER) 1;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);
//update buffer data1
SQLPutData (hstmt, data1, strlen(data2));

valuePtr = (SQLPOINTER) 0;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);
```

This attribute disables any SQLPutData function streaming that is in effect and causes each of the parameter values to be buffered on the client until the data at the SQL_DATA_AT_EXEC is closed with SQLParamData(0).

SQL_ATTR_INTERLEAVED_STREAM_PUTDATA

This attribute allows inserting LOB data with SQLParamData and SQLPutData in an interleaving fashion with function streaming. Streaming writes the LOB data directly to the connection-level buffer, bypassing the internal statement-level buffer, for improved performance.

Applications get a "Function Sequence Error (CLI0125E)" error if a new attribute is set in middle of SQLExecute, SQLParamData or SQLPutData. This error is also returned whenever there is an incorrect sequence of SQLParamData and SQLPutData.

If this attribute is set for a statement handle of a connection, any other statement handle of the same connection gets a "Function Sequence Error (CLI0125E)" if an operation is performed that uses the connection buffer until all the data has been sent to the server for that statement handle with the SQL_ATTR_INTERLEAVED_STREAM_PUTDATA attribute enabled. All SQLParamData and SQLPutData calls must be complete for the statement handle with the SQL_ATTR_INTERLEAVED_STREAM_PUTDATA attribute enabled before any other statement handle may perform an operation that uses the connection buffer. For more information about limitations when using streaming, see the “StreamPutData CLI/ODBC configuration keyword” on page 404.

To indicate the end of data for all parameters in a set of interleaved parameters, call SQLParamData with a parameter number of (0). Applications should explicitly indicate the end of data for all parameters by calling SQLParamData with a parameter number of 0.

The end of data for a single parameter is indicated by calling SQLParamData with the equivalent negative parameter number. For example, to indicate the end of the data stream for parameter number 4, the application should specify SQLParamData(-4). Applications should always indicate end of data for a parameter using the negative of the parameter number. If applications indicate the end of data for a parameter which is being streamed , CLI is able to stream data for the next parameter. This may result in better performance.

The following example shows interleaving LOB data, how to mark the end of data for a single parameter using negative parameter numbers, and how to indicate the end of data for all parameters using a parameter number of (0):

```
// Set the SQL_ATTR_INTERLEAVED_STREAM_PUTDATA attribute
SQLSetStmtAttr(hstmt, SQL_ATTR_INTERLEAVED_STREAM_PUTDATA, TRUE, 0);

//Bind the parameters with DATA_AT_EXEC indicator
blobInd = SQL_DATA_AT_EXEC;

//declare the statement handle with parameter marker
index value of 1,
//input parameter SQL_PARAM_INPUT, CLI variable type SQL_C_CHAR,
table column type CLOB,
//length of CLI variable 10, scale of decimal digits 10,
and DATA_AT_EXEC indicator
cliRC = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT,
SQL_C_CHAR, SQL_CLOB,
10, 0, &data1, 10, &blobInd);

//declare the next statement handle with
parameter marker index value of 2
cliRC = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT,
SQL_C_CHAR, SQL_CLOB,
10, 0, &data2, 10, &blobInd);

//declare the next statement handle with
parameter marker index value of 3
cliRC = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT,
SQL_C_CHAR, SQL_CLOB,
10, 0, &data3, 10, &blobInd);

SQLExecute (hstmt);

//make parameter 2 active
valuePtr = (SQLPOINTER) 2;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);

//buffer data for parameter 2
SQLPutData (hstmt, data2, strlen(data2));

//make parameter 1 active
valuePtr = (SQLPOINTER) 1;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);

// stream data for parameter 1
SQLPutData (hstmt, data1, strlen(data2));

//make parameter 2 active
valuePtr = (SQLPOINTER) 2;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);

//buffer data for parameter 2
SQLPutData (hstmt, data2, strlen(data2));

//make parameter 3 active
```

```
valuePtr = (SQLPOINTER) 3;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);

//buffer data for parameter 3
SQLPutData (hstmt, data1, strlen(data2));

//end of data for parameter 1
valuePtr = (SQLPOINTER) -1;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);

//make parameter 2 active
valuePtr = (SQLPOINTER) 2;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);

//stream the buffered data for parameter 2
SQLPutData (hstmt, data2, strlen(data2));

//make parameter 3 active valuePtr = (SQLPOINTER) 3;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);

//buffer data for parameter 3
SQLPutData (hstmt, data1, strlen(data2));

//end of data for parameter 3
valuePtr = (SQLPOINTER) -3;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);

// indicate end of data for all parameters.
// CLI streams the buffered data for all parameters
valuePtr = (SQLPOINTER) 0;
SQLParamData (hstmt, (SQLPOINTER *)&valuePtr);
```

SQL_ATTR_KEYSET_SIZE
CLI supports a pure keyset cursor, therefore the SQL_KEYSET_SIZE statement attribute is ignored. To limit the size of the keyset the application must limit the size of the result set by setting the SQL_ATTR_MAX_ROWS attribute to a value other than 0.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_LOAD_INFO
A pointer to a structure of type db2LoadStruct. The db2LoadStruct structure is used to specify all applicable LOAD options that should be used during CLI LOAD. Note that this pointer and all of its embedded pointers should be valid during every CLI function call from the time the SQL_ATTR_USE_LOAD_API statement attribute is set to the time it is turned off. For this reason, it is recommended that this pointer and its embedded pointers point to dynamically allocated memory rather than locally declared structures.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_LOAD_MODIFIED_BY
A pointer to a char string that specifies the file type modifier option to be used during CLI LOAD.

SQL_ATTR_LOAD_ROWS_COMMITTED_PTR
A pointer to an integer that represents the total number of rows processed. This value equals the number of rows successfully loaded and committed to the database, plus the number of skipped and rejected rows. The integer is 32-bit on 32-bit platforms and 64-bit on 64-bit platforms.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_LOAD_ROWS_DELETED_PTR
A pointer to an integer that represents the number of duplicate rows deleted. The integer is 32-bit on 32-bit platforms and 64-bit on 64-bit platforms.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_LOAD_ROWS_LOADED_PTR
A pointer to an integer that represents the number of rows loaded into the target table. The integer is 32-bit on 32-bit platforms and 64-bit on 64-bit platforms.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_LOAD_ROWS_READ_PTR
A pointer to an integer that represents the number of rows read. The integer is 32-bit on 32-bit platforms and 64-bit on 64-bit platforms.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_LOAD_ROWS_REJECTED_PTR
A pointer to an integer that represents the number of rows that could not be loaded. The integer is 32-bit on 32-bit platforms and 64-bit on 64-bit platforms.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_LOAD_ROWS_SKIPPED_PTR
A pointer to an integer that represents the number of rows skipped before the CLI LOAD operation began. The integer is 32-bit on 32-bit platforms and 64-bit on 64-bit platforms.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_LOB_CACHE_SIZE
A 32-bit unsigned integer that specifies maximum cache size (in bytes) for LOBs. By default, LOBs are not cached.

See the "LOBCacheSize CLI/ODBC configuration keyword" on page 367 for further usage information.

SQL_ATTR_MAX_LENGTH
A 32-bit integer value corresponding to the maximum amount of data that can be retrieved from a single character or binary column.

Note: SQL_ATTR_MAX_LENGTH should not be used to truncate data. The *BufferLength* argument of `SQLBindCol()` or `SQLGetData()` should be used instead for truncating data.
If data is truncated because the value specified for SQL_ATTR_MAX_LENGTH is less than the amount of data available, a `SQLGetData()` call or fetch returns SQL_SUCCESS instead of returning SQL_SUCCESS_WITH_INFO and SQLSTATE **01**004 (Data Truncated). The default value for SQL_ATTR_MAX_LENGTH is **0**; 0 means that CLI attempts to return all available data for character or binary type data.

SQL_ATTR_MAX_LOB_BLOCK_SIZE
A 32-bit unsigned integer that indicates the maximum size of LOB or XML data block. Specify a positive integer, up to 2 147 483 647. The default setting of 0 indicates that there is no limit to the data block size for LOB or XML data.

During data retrieval, the server includes all of the information for the current row in its reply to the client even if the maximum block size has been reached.

If both **MaxLOBBlockSize** and the **db2set** registry variable **DB2_MAX_LOB_BLOCK_SIZE** are specified, the value for **MaxLOBBlockSize** is used.

Setting the "MaxLOBBlockSize CLI/ODBC configuration keyword" on page 377 is an alternative method of specifying this behavior.

SQL_ATTR_MAX_ROWS
A 32-bit integer value corresponding to the maximum number of rows to return to the application from a query. The default value for SQL_ATTR_MAX_ROWS is **0**; 0 means all rows are returned.

SQL_ATTR_METADATA_ID
This statement attribute is defined by ODBC, but is not supported by CLI. Any attempt to set or get this attribute results in an SQLSTATE of HYC00 (Driver not capable).

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_NOSCAN
A 32-bit integer value that specifies whether CLI scans SQL strings for escape clauses. The two permitted values are:
- **SQL_NOSCAN_OFF** - SQL strings are scanned for escape clause sequences. This is the default.
- SQL_NOSCAN_ON - SQL strings are not scanned for escape clauses. Everything is sent directly to the server for processing.

This application can choose to turn off the scanning if it never uses vendor escape sequences in the SQL strings that it sends. Turning off the scanning eliminates some of the processing usage associated with the scanning.

SQL_ATTR_OPTIMIZE_FOR_NROWS
A 32-bit integer value. If it is set to an integer larger than 0, "OPTIMIZE FOR n ROWS" clause is appended to every select statement If set to 0 (the default) this clause is not appended.

The default value can also be set using the **OPTIMIZEFORNROWS** CLI/ODBC configuration keyword.

SQL_ATTR_OPTIMIZE_SQLCOLUMNS
This attribute has been deprecated.

SQL_ATTR_PARAM_BIND_OFFSET_PTR
A 32-bit integer * value that points to an offset added to pointers to change binding of dynamic parameters. If this field is non-null, CLI dereferences the pointer, adds the dereferenced value to each of the deferred fields in the descriptor record (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR), and uses the resulting pointer values at execute time. It is set to null by default.

The bind offset is always added directly to the SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR fields. If the offset is changed to a different value, the new value is added directly to the value in the descriptor field. The new offset is not added to the field value plus any earlier offsets.

Setting this statement attribute sets the SQL_DESC_BIND_OFFSET_PTR field in the APD header.

SQL_ATTR_PARAM_BIND_TYPE
A 32-bit integer value that indicates the binding orientation to be used for dynamic parameters.

This field is set to **SQL_PARAM_BIND_BY_COLUMN** (the default) to select column-wise binding.

To select row-wise binding, this field is set to the length of the structure or an instance of a buffer that is bound to a set of dynamic parameters. This length must include space for all of the bound parameters and any padding of the structure or buffer to ensure that when the address of a bound parameter is incremented with the specified length, the result points to the beginning of the same parameter in the next set of parameters. When using the sizeof operator in ANSI C, this behavior is guaranteed.

Setting this statement attribute sets the SQL_DESC_ BIND_TYPE field in the APD header.

SQL_ATTR_PARAM_OPERATION_PTR
A 16-bit unsigned integer * value that points to an array of 16-bit unsigned integer values used to specify whether or not a parameter should be ignored during execution of an SQL statement. Each value is set to either SQL_PARAM_PROCEED (for the parameter to be executed) or SQL_PARAM_IGNORE (for the parameter to be ignored).

A set of parameters can be ignored during processing by setting the status value in the array pointed to by SQL_DESC_ARRAY_STATUS_PTR in the APD to SQL_PARAM_IGNORE. A set of parameters is processed if its status value is set to SQL_PARAM_PROCEED, or if no elements in the array are set.

This statement attribute can be set to a null pointer, in which case CLI does not return parameter status values. This attribute can be set at any time, but the new value is not used until the next time `SQLExecDirect()` or `SQLExecute()` is called.

Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the APD.

SQL_ATTR_PARAM_STATUS_PTR
A 16-bit unsigned integer * value that points to an array of UWORD values containing status information for each row of parameter values after a call to `SQLExecDirect()` or `SQLExecute()`. This field is used only if SQL_ATTR_PARAMSET_SIZE is greater than 1.

The status values can contain the following values:
- SQL_PARAM_SUCCESS: The SQL statement was successfully executed for this set of parameters.
- SQL_PARAM_SUCCESS_WITH_INFO: The SQL statement was successfully executed for this set of parameters; however, warning information is available in the diagnostics data structure.
- SQL_PARAM_ERROR: There was an error in processing this set of parameters. Additional error information is available in the diagnostics data structure.
- SQL_PARAM_UNUSED: This parameter set was unused, possibly due to the fact that some previous parameter set caused an error that aborted further processing.

- SQL_PARAM_DIAG_UNAVAILABLE: CLI treats arrays of parameters as a monolithic unit and so does not generate this level of error information.

This statement attribute can be set to a null pointer, in which case CLI does not return parameter status values. This attribute can be set at any time, but the new value is not used until the next time `SQLFetch()`, `SQLFetchScroll()`, or `SQLSetPos()` is called.

Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the IPD header.

SQL_ATTR_PARAMOPT_ATOMIC
This is a 32-bit integer value which determines, when `SQLParamOptions()` has been used to specify multiple values for parameter markers, whether the underlying processing should be done via ATOMIC or NOT-ATOMIC Compound SQL. The possible values are:

- **SQL_ATOMIC_YES** - The underlying processing makes use of ATOMIC Compound SQL. This is the default if the target database supports ATOMIC compound SQL.
- SQL_ATOMIC_NO - The underlying processing makes use of NON-ATOMIC Compound SQL.

Specifying SQL_ATOMIC_YES when connected to a server that does not support ATOMIC compound SQL results in an error (SQLSTATE is **S1**C00).

Specifying SQL_ATOMIC_YES when SQL_PARC_BATCH is set to SQL_PARC_BATCH_ENABLE returns the CLI0150E error message. If you want to set SQL_PARC_BATCH to SQL_PARC_BATCH_ENABLE, you must specify SQL_ATOMIC_NO.

SQL_ATTR_PARAMS_PROCESSED_PTR
A 32-bit unsigned integer * record field that points to a buffer in which to return the current row number. As each row of parameters is processed, this is set to the number of that row. No row number is returned if this is a null pointer.

Setting this statement attribute sets the SQL_DESC_ROWS_PROCESSED_PTR field in the IPD header.

If the call to `SQLExecDirect()` or `SQLExecute()` that fills in the buffer pointed to by this attribute does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.

SQL_ATTR_PARAMSET_SIZE
A 32-bit unsigned integer value that specifies the number of values for each parameter. If SQL_ATTR_PARAMSET_SIZE is greater than 1, SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR of the APD point to arrays. The cardinality of each array is equal to the value of this field.

Setting this statement attribute sets the SQL_DESC_ARRAY_SIZE field in the APD header.

Starting in DB2 Version 9.7 Fix Pack 6, array input using SQL_ATTR_PARAMSET_SIZE, inside a trusted procedure body, is supported.

SQL_ATTR_PREFETCH
This attribute has been deprecated.

SQL_ATTR_QUERY_OPTIMIZATION_LEVEL
A 32-bit integer value that sets the query optimization level to be used on the next call to `SQLPrepare()`, `SQLExtendedPrepare()`, or `SQLExecDirect()`.

Supported values to use are: -1 (default), 0, 1, 2, 3, 5, 7, and 9.

The SQL_ATTR_QUERY_OPTIMIZATION_LEVEL statement attribute does not set the optimization level for IDS data servers. Informix optimizer directives should be used instead. For more information, see Optimizer directives

SQL_ATTR_QUERY_TIMEOUT
A 32-bit integer value that is the number of seconds to wait for an SQL statement or XQuery expression to execute before aborting the execution and returning to the application. This option can be set and used to terminate long running queries. The default value of **0** means CLI waits indefinitely for the server to complete execution of the SQL statement. CLI supports non-zero values for all platforms that support multithreading.

When using this attribute against a server which does not have native interrupt support (such as DB2 for z/OS and OS/390, Version 7 and earlier, and DB2 for i), the INTERRUPT_ENABLED option must be set when cataloging the DCS database entry for the server.

When the INTERRUPT_ENABLED option is set and this attribute is set to a non-zero value, the DB2 for i server drops the connection and rolls back the unit of work. The application receives an SQL30081N error indicating that the connection to the server has been terminated. In order for the application to process additional database requests, the application must establish a new connection with the database server.

The SQL_ATTR_QUERY_TIMEOUT can also interrupt a LOAD, which returns SQL3005N instead of SQL0952N.

SQL_ATTR_REOPT
A 32-bit integer value that enables query optimization for SQL statements that contain special registers or parameter markers. Optimization occurs by using the values available at query execution time for special registers or parameter markers, instead of the default estimates that are chosen by the compiler. The valid values of the attribute are:

- 2 = SQL_REOPT_NONE. This is the default. No query optimization occurs at query execution time. The default estimates chosen by the compiler are used for the special registers or parameter markers. The default NULLID package set is used to execute dynamic SQL statements.
- 3 = SQL_REOPT_ONCE. Query optimization occurs once at query execution time, when the query is executed for the first time. The NULLIDR1 package set, which is bound with the REOPT ONCE bind option, is used.
- 4 = SQL_REOPT_ALWAYS. Query optimization or reoptimization occurs at query execution time every time the query is executed. The NULLIDRA package set, which is bound with the REOPT ALWAYS bind option, is used.

The NULLIDR1 and NULLIDRA are reserved package set names, and when used, REOPT ONCE and REOPT ALWAYS are implied. These package sets have to be explicitly created with these commands:

```
db2 bind db2clipk.bnd collection NULLIDR1
db2 bind db2clipk.bnd collection NULLIDRA
```

SQL_ATTR_REOPT and SQL_ATTR_CURRENT_PACKAGE_SET are mutually exclusive, therefore, if one is set, the other is not allowed.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_RETRIEVE_DATA
A 32-bit integer value:

- **SQL_RD_ON** = `SQLFetchScroll()` and in DB2 CLI v5 and later, `SQLFetch()`, retrieve data after it positions the cursor to the specified location. This is the default.
- SQL_RD_OFF = `SQLFetchScroll()` and in DB2 CLI v5 and later, `SQLFetch()`, do not retrieve data after it positions the cursor.

By setting SQL_RETRIEVE_DATA to SQL_RD_OFF, an application can verify if a row exists or retrieve a bookmark for the row without incurring increased resource usage associated with retrieving rows.

SQL_ATTR_RETURN_USER_DEFINED_TYPES
A Boolean attribute that specifies whether user-defined type columns are reported as the user-defined type or the underlying base type when queried by functions such as `SQLDescribeCol()`. The default value is 0 (false), where columns are reported as the underlying base type.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_ROW_ARRAY_SIZE
A 32-bit integer value that specifies the number of rows in the rowset. This is the number of rows returned by each call to `SQLFetch()` or `SQLFetchScroll()`. The default value is 1.

If the specified rowset size exceeds the maximum rowset size supported by the data source, CLI substitutes that value and returns SQLSTATE 01S02 (Option value changed).

This option can be specified for an open cursor.

Setting this statement attribute sets the SQL_DESC_ARRAY_SIZE field in the ARD header.

SQL_ATTR_ROW_BIND_OFFSET_PTR
A 32-bit integer * value that points to an offset added to pointers to change binding of column data. If this field is non-null, CLI dereferences the pointer, adds the dereferenced value to each of the deferred fields in the descriptor record (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR), and uses the new pointer values when binding. It is set to null by default.

Setting this statement attribute sets the SQL_DESC_BIND_OFFSET_PTR field in the ARD header.

SQL_ATTR_ROW_BIND_TYPE
A 32-bit integer value that sets the binding orientation to be used when `SQLFetch()` or `SQLFetchScroll()` is called on the associated statement. Column-wise binding is selected by supplying the defined constant SQL_BIND_BY_COLUMN in **ValuePtr*. Row-wise binding is selected by supplying a value in **ValuePtr* specifying the length of a structure or an instance of a buffer into which result columns are bound.

The length specified in **ValuePtr* must include space for all of the bound columns and any padding of the structure or buffer to ensure that when the address of a bound column is incremented with the specified length,

the result points to the beginning of the same column in the next row. When using the **sizeof** operator with structures or unions in ANSI C, this behavior is guaranteed.

Column-wise binding is the default binding orientation for `SQLFetch()` and `SQLFetchScroll()`.

Setting this statement attribute sets the SQL_DESC_BIND_TYPE field in the ARD header.

SQL_ATTR_ROW_NUMBER
A 32-bit integer value that is the number of the current row in the entire result set. If the number of the current row cannot be determined or there is no current row, CLI returns 0.

This attribute can be retrieved by a call to `SQLGetStmtAttr()`, but not set by a call to `SQLSetStmtAttr()`.

SQL_ATTR_ROW_OPERATION_PTR
A 16-bit unsigned integer * value that points to an array of UDWORD values used to ignore a row during a bulk operation using `SQLSetPos()`. Each value is set to either SQL_ROW_PROCEED (for the row to be included in the bulk operation) or SQL_ROW_IGNORE (for the row to be excluded from the bulk operation).

This statement attribute can be set to a null pointer, in which case CLI does not return row status values. This attribute can be set at any time, but the new value is not used until the next time `SQLFetch()`, `SQLFetchScroll()`, or `SQLSetPos()` is called.

Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the ARD.

SQL_ATTR_ROW_STATUS_PTR
A 16-bit unsigned integer * value that points to an array of UWORD values containing row status values after a call to `SQLFetch()` or `SQLFetchScroll()`. The array has as many elements as there are rows in the rowset.

This statement attribute can be set to a null pointer, in which case CLI does not return row status values. This attribute can be set at any time, but the new value is not used until the next time `SQLFetch()`, `SQLFetchScroll()`, or `SQLSetPos()` is called.

Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the IRD header.

SQL_ATTR_ROWS_FETCHED_PTR
A 32-bit unsigned integer * value that points to a buffer in which to return the number of rows fetched after a call to `SQLFetch()` or `SQLFetchScroll()`.

Setting this statement attribute sets the SQL_DESC_ROWS_PROCESSED_PTR field in the IRD header.

This attribute is mapped by CLI to the *RowCountPtr* array in a call to `SQLExtendedFetch()`.

SQL_ATTR_ROWCOUNT_PREFETCH
This attribute enables CLI to determine the number of rows so that the entire result set can be prefetched. This attribute has one of the following values:

- **0 (default)**: Off

- **1**: On

If you set SQL_ATTR_ROWCOUNT_PREFETCH to 0 and call SQLRowCount() using a non-scrollable SELECT-only cursor, the function sets the contents of *RowCountPtr* to -1 because the number of rows is not available until all of the data has been fetched.

If you set SQL_ATTR_ROWCOUNT_PREFETCH to 1 and call SQLRowCount() using a non-scrollable SELECT-only cursor, the following behaviors are observed:

- If you use SELECT * FROM INSERT | UPDATE | DELETE statements with forward-only cursors, the row count comes from the SELECT statements. This is different than the rows-affected count that is provided with these cursors without the SQL_ATTR_ROWCOUNT_PREFETCH attribute set.
- All cursor data is prefetched. This might take several round trips to the server and a considerable amount of memory on the client.
- The prefetched data is not discarded; instead, it is used to satisfy the fetch requests by the application.

This attribute is not applicable to scrollable cursors because they can provide a row count.

Specify this attribute before preparing a statement.

Restriction: SQL_ATTR_ROWCOUNT_PREFETCH is not supported when the cursor contains LOBs or XML.

SQL_ROWSET_SIZE

CLI applications should now use `SQLFetchScroll()` rather than `SQLExtendedFetch()`. Applications should also use the statement attribute SQL_ATTR_ROW_ARRAY_SIZE to set the number of rows in the rowset.

A 32-bit integer value that specifies the number of rows in the rowset. A rowset is the array of rows returned by each call to `SQLExtendedFetch()`. The default value is **1**, which is equivalent to making a single `SQLFetch()` call. This option can be specified even when the cursor is open and becomes effective on the next `SQLExtendedFetch()` call.

SQL_ATTR_SIMULATE_CURSOR (ODBC 2.0)

This statement attribute is not supported by CLI but is defined by ODBC.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_STMT_CONCENTRATOR

Specifies whether dynamic statements that contain literal values use the statement cache.

- SQL_STMT_CONCENTRATOR_OFF - The statement concentrator behavior is disabled.
- SQL_STMT_CONCENTRATOR_WITH_LITERALS - The statement concentrator with literal behavior is enabled for situations that are supported by the server. For example, the statement concentrator is not enabled if the statement has parameter markers, named parameter markers, or a mix of literals, parameter markers, and named parameter markers.

Setting the "StmtConcentrator CLI/ODBC configuration keyword" on page 403 is an alternative method of specifying this behavior.

Note: When this attribute is used against DB2® for z/OS® servers older than Version 10, the request is ignored.

SQL_ATTR_STMTTXN_ISOLATION
See SQL_ATTR_TXN_ISOLATION.

SQL_ATTR_STREAM_GETDATA
A 32-bit unsigned integer that indicates if the data output stream for the `SQLGetData()` function is optimized. The values are:

- **0 (default)**: CLI buffers all the data on the client.
- 1: For applications that do not need to buffer data and are querying data on a server that supports Dynamic Data Format, specify 1 to indicate that data buffering is not required. The CLI client optimizes the data output stream.

This keyword is ignored if Dynamic Data Format is not supported by the server.

If **`StreamGetData`** is set to 1 and CLI cannot determine the number of bytes still available to return in the output buffer, `SQLGetData()` returns SQL_NO_TOTAL (-4) as the length when truncation occurs. Otherwise, `SQLGetData()` returns the number of bytes still available.

Setting the "StreamGetData CLI/ODBC configuration keyword" on page 404 is an alternative method of specifying this behavior.

SQL_ATTR_TXN_ISOLATION
A 32-bit integer value that sets the transaction isolation level for the current *StatementHandle*.

This option cannot be set if there is an open cursor on this statement handle (SQLSTATE **24**000).

The value SQL_ATTR_STMTTXN_ISOLATION is synonymous with SQL_ATTR_TXN_ISOLATION. However, since the ODBC Driver Manager rejects the setting of SQL_ATTR_TXN_ISOLATION as a statement option, ODBC applications that need to set translation isolation level on a per statement basis must use the manifest constant SQL_ATTR_STMTTXN_ISOLATION instead on the `SQLSetStmtAttr()` call.

The default transaction isolation level can also be set using the **`TXNISOLATION`** CLI/ODBC configuration keyword.

This attribute (or corresponding keyword) is only applicable if the default isolation level is used for the statement handle. If the application has specifically set the isolation level for the statement handle, then this attribute does not have effect.

Note: It is an IBM extension to allow setting this option at the statement level.

SQL_ATTR_USE_BOOKMARKS
A 32-bit integer value that specifies whether an application CLI0150E is returned use bookmarks with a cursor:

- **SQL_UB_OFF** = Off (the default)
- SQL_UB_VARIABLE = An application will use bookmarks with a cursor, and CLI provides variable-length bookmarks if they are supported.

To use bookmarks with a cursor, the application must specify this option with the SQL_UB_VARIABLE value before opening the cursor.

SQL_ATTR_USE_LOAD_API
A 32-bit integer that indicates if the LOAD utility replaces the regular CLI array insert for inserting data. The possible values are:

SQL_USE_LOAD_OFF
(Default) Use regular CLI array insert to insert data.

SQL_USE_LOAD_INSERT
Use the LOAD utility to append to existing data in the table.

SQL_USE_LOAD_REPLACE
Use the LOAD utility to replace existing data in the table.

SQL_USE_LOAD_RESTART
Resume a previously failed CLI LOAD operation. If the previous CLI LOAD operation failed while rows were being inserted (that is, before the SQL_ATTR_USE_LOAD_API statement attribute was set to SQL_USE_LOAD_OFF), the CLI LOAD feature remains active, and subsequent rows are inserted by the CLI LOAD utility. Otherwise, if the operation failed while CLI LOAD was being turned off, regular CLI array inserts resume after the restarted load completes.

SQL_USE_LOAD_TERMINATE
Clean up and undo a previously failed CLI LOAD operation. After setting the statement attribute to this value, regular CLI array inserts will resume.

This attribute is not supported when accessing IDS data servers.

Note: Starting from Version 9.7, Fix Pack 4, this attribute can be used with SQL_ATTR_ASYNC_ENABLE.

SQL_ATTR_XML_DECLARATION
A 32-bit unsigned integer that specifies which elements of an XML declaration are added to XML data when it is implicitly serialized. This attribute does not affect the result of the XMLSERIALIZE function.

This attribute can only be specified on a statement handle that has no open cursors associated with it. Attempting to update the value of this attribute while there are open cursors on the statement handle results in a CLI0126E (SQLSTATE HY011) error, and the value remains unchanged.

Set this attribute to the sum of each component required:

0 No declarations or byte order marks (BOMs) are added to the output buffer.

1 A byte order mark (BOM) in the appropriate endianness is prepended to the output buffer if the target encoding is UTF-16 or UTF-32. (Although a UTF-8 BOM exists, DB2 does not generate it, even if the target encoding is UTF-8.)

2 A minimal XML declaration is generated, containing only the XML version.

4 An encoding attribute that identifies the target encoding is added to any generated XML declaration. Therefore, this setting only has effect when the setting of 2 is also included when computing the value of this attribute.

Attempts to set any other value using SQLSetStmtAttr() or SQLSetStmtOption() results in a CLI0191E (SQLSTATE HY024) error, and the value remains unchanged.

The default setting is 7, which indicates that a BOM and an XML declaration containing the XML version and encoding attribute are generated during implicit serialization.

This attribute can also be specified on a connection handle and affects any statement handles allocated after the value is changed. Existing statement handles retain their original values.

This attribute is not supported when accessing IDS data servers.

SQL_ATTR_XQUERY_STATEMENT

A 32-bit integer value that specifies whether the statement associated with the current statement handle is an XQuery expression or an SQL statement or query. This can be used by CLI applications that do not want to prefix an XQuery expression with the "XQUERY" keyword. The supported values are:

SQL_TRUE
The next statement executed on the current statement handle is processed as an XQuery expression. If the server does not support XQuery, setting this attribute to SQL_TRUE results in a warning, CLI0005W (SQLSTATE 01S02), and the attribute's value is unchanged.

SQL_FALSE (default)
The next statement executed on the current statement handle is processed as an SQL statement.

This attribute takes effect on the next SQLPrepare() or SQLExecDirect() function call.

This attribute is not supported when accessing IDS data servers.

Chapter 5. Descriptor values

Descriptor FieldIdentifier argument values (CLI)

The *FieldIdentifier* argument indicates the descriptor field to be set. A descriptor contains the descriptor header, consisting of the header fields described in the next section, and zero or more descriptor records, consisting of the record fields described in the following section.

Header fields

Each descriptor has a header consisting of the following fields.

SQL_DESC_ALLOC_TYPE [All] This read-only SQLSMALLINT header field specifies whether the descriptor was allocated automatically by CLI or explicitly by the application. The application can obtain, but not modify, this field. The field is set to SQL_DESC_ALLOC_AUTO if the descriptor was automatically allocated. It is set to SQL_DESC_ALLOC_USER if the descriptor was explicitly allocated by the application.

SQL_DESC_ARRAY_SIZE [Application descriptors] In ARDs, this SQLUINTEGER header field specifies the number of rows in the rowset. This is the number of rows to be returned by a call to `SQLFetch()`, `SQLFetchScroll()`, or `SQLSetPos()`. The default value is 1. This field can also be set by calling `SQLSetStmtAttr()` with the SQL_ATTR_ROW_ARRAY_SIZE attribute.

In APDs, this SQLUINTEGER header field specifies the number of values for each parameter.

The default value of this field is 1. If SQL_DESC_ARRAY_SIZE is greater than 1, SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR of the APD or ARD point to arrays. The cardinality of each array is equal to the value of this field.

This field in the ARD can also be set by calling `SQLSetStmtAttr()` with the SQL_ROWSET_SIZE attribute. This field in the APD can also be set by calling `SQLSetStmtAttr()` with the SQL_ATTR_PARAMSET_SIZE attribute.

SQL_DESC_ARRAY_STATUS_PTR [All] For each descriptor type, this SQLUSMALLINT * header field points to an array of SQLUSMALLINT values. This array is referred to as:
- row status array (IRD)
- parameter status array (IPD)
- row operation array (ARD)
- parameter operation array (APD)

In the IRD, this header field points to a row status array containing status values after a call to `SQLFetch()`, `SQLFetchScroll()`, or `SQLSetPos()`. The array has as many elements as there are rows in the rowset. The application must allocate an array of SQLUSMALLINTs and set this field to point to the array. The field is set to a null pointer by default. CLI will populate the array, unless the SQL_DESC_ARRAY_STATUS_PTR field is set to a null pointer, in which case no status values are generated and the array is not populated.

Note: Behavior is undefined if the application sets the elements of the row status array pointed to by the SQL_DESC_ARRAY_STATUS_PTR field of the IRD. The array is initially populated by a call to `SQLFetch()`, `SQLFetchScroll()`, or `SQLSetPos()`. If the call did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the array pointed to by this field are undefined.

The elements in the array can contain the following values:

- SQL_ROW_SUCCESS: The row was successfully fetched and has not changed since it was last fetched.
- SQL_ROW_SUCCESS_WITH_INFO: The row was successfully fetched and has not changed since it was last fetched. However, a warning was returned about the row.
- SQL_ROW_ERROR: An error occurred while fetching the row.
- SQL_ROW_UPDATED: The row was successfully fetched and has been updated since it was last fetched. If the row is fetched again, its status is SQL_ROW_SUCCESS.
- SQL_ROW_DELETED: The row has been deleted since it was last fetched.
- SQL_ROW_ADDED: The row was inserted by `SQLSetPos()`. If the row is fetched again, its status is SQL_ROW_SUCCESS.
- SQL_ROW_NOROW: The rowset overlapped the end of the result set and no row was returned that corresponded to this element of the row status array.

This field in the ARD can also be set by calling `SQLSetStmtAttr()` with the SQL_ATTR_ROW_STATUS_PTR attribute.

In the IPD, this header field points to a parameter status array containing status information for each set of parameter values after a call to `SQLExecute()` or `SQLExecDirect()`. If the call to `SQLExecute()` or `SQLExecDirect()` did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the array pointed to by this field are undefined. The application must allocate an array of SQLUSMALLINTs and set this field to point to the array. The driver will populate the array, unless the SQL_DESC_ARRAY_STATUS_PTR field is set to a null pointer, in which case no status values are generated and the array is not populated.

The elements in the array can contain the following values:

- SQL_PARAM_SUCCESS: The SQL statement was successfully executed for this set of parameters.
- SQL_PARAM_SUCCESS_WITH_INFO: The SQL statement was successfully executed for this set of parameters; however, warning information is available in the diagnostics data structure.
- SQL_PARAM_ERROR: An error occurred in processing this set of parameters. Additional error information is available in the diagnostics data structure.
- SQL_PARAM_UNUSED: This parameter set was unused, possibly due to the fact that some previous parameter set caused an error that aborted further processing.
- SQL_PARAM_DIAG_UNAVAILABLE: Diagnostic information is not available. An example of this is when CLI treats arrays of parameters as a monolithic unit and so does not generate this level of error information.

This field in the APD can also be set by calling `SQLSetStmtAttr()` with the SQL_ATTR_PARAM_STATUS_PTR attribute.

In the ARD, this header field points to a row operation array of values that can be set by the application to indicate whether this row is to be ignored for SQLSetPos() operations.

The elements in the array can contain the following values:

- SQL_ROW_PROCEED: The row is included in the bulk operation using SQLSetPos(). (This setting does not guarantee that the operation will occur on the row. If the row has the status SQL_ROW_ERROR in the IRD row status array, CLI may not be able to perform the operation in the row.)
- SQL_ROW_IGNORE: The row is excluded from the bulk operation using SQLSetPos().

If no elements of the array are set, all rows are included in the bulk operation. If the value in the SQL_DESC_ARRAY_STATUS_PTR field of the ARD is a null pointer, all rows are included in the bulk operation; the interpretation is the same as if the pointer pointed to a valid array and all elements of the array were SQL_ROW_PROCEED. If an element in the array is set to SQL_ROW_IGNORE, the value in the row status array for the ignored row is not changed.

This field in the ARD can also be set by calling SQLSetStmtAttr() with the SQL_ATTR_ROW_OPERATION_PTR attribute.

In the APD, this header field points to a parameter operation array of values that can be set by the application to indicate whether this set of parameters is to be ignored when SQLExecute() or SQLExecDirect() is called. The elements in the array can contain the following values:

- SQL_PARAM_PROCEED: The set of parameters is included in the SQLExecute() or SQLExecDirect() call.
- SQL_PARAM_IGNORE: The set of parameters is excluded from the SQLExecute() or SQLExecDirect() call.

If no elements of the array are set, all sets of parameters in the array are used in the SQLExecute() or SQLExecDirect() calls. If the value in the SQL_DESC_ARRAY_STATUS_PTR field of the APD is a null pointer, all sets of parameters are used; the interpretation is the same as if the pointer pointed to a valid array and all elements of the array were SQL_PARAM_PROCEED.

This field in the APD can also be set by calling SQLSetStmtAttr() with the SQL_ATTR_PARAM_OPERATION_PTR attribute.

SQL_DESC_BIND_OFFSET_PTR [Application descriptors] This SQLINTEGER * header field points to the bind offset. It is set to a null pointer by default. If this field is not a null pointer, CLI dereferences the pointer and adds the dereferenced value to each of the deferred fields that has a non-null value in the descriptor record (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR) at fetch time, and uses the new pointer values when binding.

The bind offset is always added directly to the values in the SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR fields. If the offset is changed to a different value, the new value is still added directly to the value in each descriptor field. The new offset is not added to the field value plus any earlier offset.

This field is a *deferred field*: it is not used at the time it is set, but is used at a later time by CLI to retrieve data.

This field in the ARD can also be set by calling `SQLSetStmtAttr()` with the SQL_ATTR_ROW_BIND_OFFSET_PTR attribute. This field in the ARD can also be set by calling `SQLSetStmtAttr()` with the SQL_ATTR_PARAM_BIND_OFFSET_PTR attribute.

SQL_DESC_BIND_TYPE [Application descriptors] This SQLINTEGER header field sets the binding orientation to be used for either binding columns or parameters.

In ARDs, this field specifies the binding orientation when `SQLFetchScroll()` is called on the associated statement handle.

To select column-wise binding for columns, this field is set to SQL_BIND_BY_COLUMN (the default).

This field in the ARD can also be set by calling `SQLSetStmtAttr()` with SQL_ATTR_ROW_BIND_TYPE Attribute.

In APDs, this field specifies the binding orientation to be used for dynamic parameters.

To select column-wise binding for parameters, this field is set to SQL_BIND_BY_COLUMN (the default).

This field in the APD can also be set by calling `SQLSetStmtAttr()` with SQL_ATTR_PARAM_BIND_TYPE Attribute.

SQL_DESC_COUNT [All] This SQLSMALLINT header field specifies the one-based index of the highest-numbered record that contains data. When CLI sets the data structure for the descriptor, it must also set the COUNT field to show how many records are significant. When an application allocates an instance of this data structure, it does not have to specify how many records to reserve room for. As the application specifies the contents of the records, CLI takes any required action to ensure that the descriptor handle refers to a data structure of adequate size.

SQL_DESC_COUNT is not a count of all data columns that are bound (if the field is in an ARD), or all parameters that are bound (in an APD), but the number of the highest-numbered record. If a column or a parameter with a number that is less than the number of the highest-numbered column is unbound (by calling `SQLBindCol()` with the *Target ValuePtr* argument set to a null pointer, or `SQLBindParameter()` with the *Parameter ValuePtr* argument set to a null pointer), SQL_DESC_COUNT is not changed. If additional columns or parameters are bound with numbers greater than the highest-numbered record that contains data, CLI automatically increases the value in the SQL_DESC_COUNT field. If all columns or parameters are unbound by calling `SQLFreeStmt()` with the SQL_UNBIND option, SQL_DESC_COUNT is set to 0.

The value in SQL_DESC_COUNT can be set explicitly by an application by calling `SQLSetDescField()`. If the value in SQL_DESC_COUNT is explicitly decreased, all records with numbers greater than the new value in SQL_DESC_COUNT are removed, unbinding the columns. If the value in SQL_DESC_COUNT is explicitly set to 0, and the field is in an APD, all parameters are unbound. If the value in

SQL_DESC_COUNT is explicitly set to 0, and the field is in an ARD, all data buffers except a bound bookmark column are released.

The record count in this field of an ARD does not include a bound bookmark column.

SQL_DESC_ROWS_PROCESSED_PTR [Implementation descriptors] In an IRD, this SQLUINTEGER * header field points to a buffer containing the number of rows fetched after a call to `SQLFetch()` or `SQLFetchScroll()`, or the number of rows affected in a bulk operation performed by a call to `SQLSetPos()`.

In an IPD, this SQLUINTEGER * header field points to a buffer containing the number of the row as each row of parameters is processed. No row number will be returned if this is a null pointer.

SQL_DESC_ROWS_PROCESSED_PTR is valid only after SQL_SUCCESS or SQL_SUCCESS_WITH_INFO has been returned after a call to `SQLFetch()` or `SQLFetchScroll()` (for an IRD field) or `SQLExecute()` or `SQLExecDirect()` (for an IPD field). If the return code is not SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the location pointed to by SQL_DESC_ROWS_PROCESSED_PTR is undefined. If the call that fills in the buffer pointed to by this field did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined, unless it returns SQL_NO_DATA, in which case the value in the buffer is set to 0.

This field in the ARD can also be set by calling `SQLSetStmtAttr()` with the SQL_ATTR_ROWS_FETCHED_PTR attribute. This field in the ARD can also be set by calling `SQLSetStmtAttr()` with the SQL_ATTR_PARAMS_PROCESSED_PTR attribute.

The buffer pointed to by this field is allocated by the application. It is a deferred output buffer that is set by CLI. It is set to a null pointer by default.

Record fields

Each descriptor contains one or more records consisting of fields that define either column data or dynamic parameters, depending on the type of descriptor. Each record is a complete definition of a single column or parameter.

SQL_DESC_AUTO_UNIQUE_VALUE [IRDs] This read-only SQLINTEGER record field contains SQL_TRUE if the column is an auto-incrementing column, or SQL_FALSE if the column is not an auto-incrementing column. This field is read-only, but the underlying auto-incrementing column is not necessarily read-only.

SQL_DESC_BASE_COLUMN_NAME [IRDs] This read-only SQLCHAR record field contains the base column name for the result set column. If a base column name does not exist (as in the case of columns that are expressions), then this variable contains an empty string.

SQL_DESC_BASE_TABLE_NAME [IRDs] This read-only SQLCHAR record field contains the base table name for the result set column. If a base table name cannot be defined or is not applicable, then this variable contains an empty string.

SQL_DESC_CASE_SENSITIVE [Implementation descriptors] This read-only SQLINTEGER record field contains SQL_TRUE if the column or parameter is

treated as case-sensitive for collations and comparisons, or SQL_FALSE if the column is not treated as case-sensitive for collations and comparisons, or if it is a non-character column.

SQL_DESC_CATALOG_NAME [IRDs] This read-only SQLCHAR record field contains the catalog or qualifier name for the base table that contains the column. The return value is driver-dependent if the column is an expression or if the column is part of a view. If the data source does not support catalogs (or qualifiers) or the catalog or qualifier name cannot be determined, this variable contains an empty string.

SQL_DESC_CONCISE_TYPE [All] This SQLSMALLINT header field specifies the concise data type for all data types.

The values in the SQL_DESC_CONCISE_TYPE and SQL_DESC_TYPE fields are interdependent. Each time one of the fields is set, the other must also be set. SQL_DESC_CONCISE_TYPE can be set by a call to `SQLBindCol()` or `SQLBindParameter()`, or `SQLSetDescField()`. SQL_DESC_TYPE can be set by a call to `SQLSetDescField()` or `SQLSetDescRec()`.

If SQL_DESC_CONCISE_TYPE is set to a concise data type, SQL_DESC_TYPE field is set to the same value, and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to 0.

SQL_DESC_DATA_PTR [Application descriptors and IPDs] This SQLPOINTER record field points to a variable that will contain the parameter value (for APDs) or the column value (for ARDs). The descriptor record (and either the column or parameter that it represents) is unbound if Target*ValuePtr* in a call to either `SQLBindCol()` or `SQLBindParameter()` is a null pointer, or the SQL_DESC_DATA_PTR field in a call to `SQLSetDescField()` or `SQLSetDescRec()` is set to a null pointer. Other fields are not affected if the SQL_DESC_DATA_PTR field is set to a null pointer. If the call to `SQLFetch()` or `SQLFetchScroll()` that fills in the buffer pointed to by this field did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.

This field is a deferred field: it is not used at the time it is set, but is used at a later time by CLI to retrieve data.

Whenever the SQL_DESC_DATA_PTR field is set, CLI checks that the value in the SQL_DESC_TYPE field contains valid CLI or ODBC data types, and that all other fields affecting the data types are consistent. Refer to the consistency checks information for more detail.

SQL_DESC_DATETIME_INTERVAL_CODE [All] This SQLSMALLINT record field contains the subcode for the specific datetime data type when the SQL_DESC_TYPE field is SQL_DATETIME. This is true for both SQL and C data types.

This field can be set to the following for datetime data types:

Table 169. Datetime subcodes

Datetime types	DATETIME_INTERVAL_CODE
SQL_TYPE_DATE/SQL_C_TYPE_DATE	SQL_CODE_DATE
SQL_TYPE_TIME/SQL_C_TYPE_TIME	SQL_CODE_TIME

Table 169. Datetime subcodes (continued)

Datetime types	DATETIME_INTERVAL_CODE
	SQL_CODE_TIMESTAMP
SQL_TYPE_TIMESTAMP/ SQL_C_TYPE_TIMESTAMP	

ODBC 3.0 defines other values (not listed here) for interval data types, which CLI does not support. If any other value is specified in a `SQLSetDescRec()` or `SQLSetDescField()` call, an error will be generated indicating HY092 (Option type out of range).

SQL_DESC_DATETIME_INTERVAL_PRECISION [All] ODBC 3.0 defines this SQLINTEGER record field, however, CLI does not support interval data types. The fixed value returned is 0. Any attempt to set this field will result in 01S02 (Option value changed).

SQL_DESC_DISPLAY_SIZE [IRDs] This read-only SQLINTEGER record field contains the maximum number of characters required to display the data from the column. The value in this field is not the same as the descriptor field SQL_DESC_LENGTH because the LENGTH field is undefined for all numeric types.

SQL_DESC_FIXED_PREC_SCALE [Implementation descriptors] This read-only SQLSMALLINT record field is set to SQL_TRUE if the column is an exact numeric column and has a fixed precision and non-zero scale, or SQL_FALSE if the column is not an exact numeric column with a fixed precision and scale.

SQL_DESC_INDICATOR_PTR [Application descriptors] In ARDs, this SQLINTEGER * record field points to the indicator variable. This variable contains SQL_NULL_DATA if the column value is NULL. For APDs, the indicator variable is set to SQL_NULL_DATA to specify NULL dynamic arguments. Otherwise, the variable is zero (unless the values in SQL_DESC_INDICATOR_PTR and SQL_DESC_OCTET_LENGTH_PTR are the same pointer).

If the SQL_DESC_INDICATOR_PTR field in an ARD is a null pointer, CLI is prevented from returning information about whether the column is NULL or not. If the column is NULL and INDICATOR_PTR is a null pointer, SQLSTATE 22002, Indicator variable required but not supplied, is returned when CLI attempts to populate the buffer after a call to `SQLFetch()` or `SQLFetchScroll()`. If the call to `SQLFetch()` or `SQLFetchScroll()` did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.

The SQL_DESC_INDICATOR_PTR field determines whether the field pointed to by SQL_DESC_OCTET_LENGTH_PTR is set. If the data value for a column is NULL, CLI sets the indicator variable to SQL_NULL_DATA. The field pointed to by SQL_DESC_OCTET_LENGTH_PTR is then not set. If a NULL value is not encountered during the fetch, the buffer pointed to by SQL_DESC_INDICATOR_PTR is set to zero, and the buffer pointed to by SQL_DESC_OCTET_LENGTH_PTR is set to the length of the data.

If the INDICATOR_PTR field in an APD is a null pointer, the application cannot use this descriptor record to specify NULL arguments.

This field is a deferred field: it is not used at the time it is set, but is used at a later time by CLI to store data.

SQL_DESC_LABEL [IRDs] This read-only SQLCHAR record field contains the column label or title. If the column does not have a label, this variable contains the column name. If the column is unnamed and unlabeled, this variable contains an empty string.

SQL_DESC_LENGTH [All] This SQLUINTEGER record field is either the maximum or actual character length of a character string or a binary data type. It is the maximum character length for a fixed-length data type, or the actual character length for a variable-length data type. Its value always excludes the null termination character that ends the character string. Note that this field is a count of characters, not a count of bytes.

SQL_DESC_LITERAL_PREFIX [IRDs] This read-only SQLCHAR record field contains the character or characters that CLI recognizes as a prefix for a literal of this data type. This variable contains an empty string for a data type for which a literal prefix is not applicable.

SQL_DESC_LITERAL_SUFFIX [IRDs] This read-only SQLCHAR record field contains the character or characters that CLI recognizes as a suffix for a literal of this data type. This variable contains an empty string for a data type for which a literal suffix is not applicable.

SQL_DESC_LOCAL_TYPE_NAME [Implementation descriptors] This read-only SQLCHAR record field contains any localized (native language) name for the data type that may be different from the regular name of the data type. If there is no localized name, then an empty string is returned. This field is for display purposes only.

SQL_DESC_NAME [Implementation descriptors] This SQLCHAR record field in a row descriptor contains the column alias, if it applies. If the column alias does not apply, the column name is returned. In either case, the UNNAMED field is set to SQL_NAMED. If there is no column name or a column alias, an empty string is returned in the NAME field and the UNNAMED field is set to SQL_UNNAMED.

An application can set the SQL_DESC_NAME field of an IPD to a parameter name or alias to specify stored procedure parameters by name. The SQL_DESC_NAME field of an IRD is a read-only field; SQLSTATE HY091 (Invalid descriptor field identifier) will be returned if an application attempts to set it.

In IPDs, this field is undefined if dynamic parameters are not supported. If named parameters are supported and the version of CLI is capable of describing parameters, then the parameter name is returned in this field.

The column name value can be affected by the environment attribute SQL_ATTR_USE_LIGHT_OUTPUT_SQLDA set by `SQLSetEnvAttr()`.

SQL_DESC_NULLABLE [Implementation descriptors] In IRDs, this read-only SQLSMALLINT record field is SQL_NULLABLE if the column can have NULL values; SQL_NO_NULLS if the column cannot have NULL values; or SQL_NULLABLE_UNKNOWN if it is not known whether the column accepts NULL values. This field pertains to the result set column, not the base column.

In IPDs, this field is always set to SQL_NULLABLE, since dynamic parameters are always nullable, and cannot be set by an application.

SQL_DESC_NUM_PREC_RADIX [All] This SQLINTEGER field contains a value of 2 if the data type in the SQL_DESC_TYPE field is an approximate numeric data type, because the SQL_DESC_PRECISION field contains the number of bits. This field contains a value of 10 if the data type in the SQL_DESC_TYPE field is an exact numeric data type, because the SQL_DESC_PRECISION field contains the number of decimal digits. This field is set to 0 for all non-numeric data types.

SQL_DESC_OCTET_LENGTH [All] This SQLINTEGER record field contains the length, in bytes, of a character string or binary data type. For fixed-length character types, this is the actual length in bytes. For variable-length character or binary types, this is the maximum length in bytes. This value always excludes space for the null termination character for implementation descriptors and always includes space for the null termination character for application descriptors. For application data, this field contains the size of the buffer. For APDs, this field is defined only for output or input/output parameters.

SQL_DESC_OCTET_LENGTH_PTR [Application descriptors] This SQLINTEGER * record field points to a variable that will contain the total length in bytes of a dynamic argument (for parameter descriptors) or of a bound column value (for row descriptors).

For an APD, this value is ignored for all arguments except character string and binary; if this field points to SQL_NTS, the dynamic argument must be null-terminated. To indicate that a bound parameter will be a data-at-execute parameter, an application sets this field in the appropriate record of the APD to a variable that, at execute time, will contain the value SQL_DATA_AT_EXEC. If there is more than one such field, SQL_DESC_DATA_PTR can be set to a value uniquely identifying the parameter to help the application determine which parameter is being requested.

If the OCTET_LENGTH_PTR field of an ARD is a null pointer, CLI does not return length information for the column. If the SQL_DESC_OCTET_LENGTH_PTR field of an APD is a null pointer, CLI assumes that character strings and binary values are null terminated. (Binary values should not be null terminated, but should be given a length, in order to avoid truncation.)

If the call to `SQLFetch()` or `SQLFetchScroll()` that fills in the buffer pointed to by this field did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.

This field is a deferred field: it is not used at the time it is set, but is used at a later time by CLI to buffer data.

By default this is a pointer to a 4-byte value.

SQL_DESC_PARAMETER_TYPE [IPDs] This SQLSMALLINT record field is set to SQL_PARAM_INPUT for an input parameter, SQL_PARAM_INPUT_OUTPUT for an input/output parameter, or SQL_PARAM_OUTPUT for an output parameter. Set to SQL_PARAM_INPUT by default.

For an IPD, the field is set to SQL_PARAM_INPUT by default if the IPD is not automatically populated by CLI (the SQL_ATTR_ENABLE_AUTO_IPD statement attribute is SQL_FALSE). An application should set this field in the IPD for parameters that are not input parameters.

SQL_DESC_PRECISION [All] This SQLSMALLINT record field contains the number of digits for an exact numeric type, the number of bits in the mantissa (binary precision) for an approximate numeric type, or the numbers of digits in the fractional seconds component for the SQL_TYPE_TIME or SQL_TYPE_TIMESTAMP data types. This field is undefined for all other data types.

SQL_DESC_SCALE [All] This SQLSMALLINT record field contains the defined scale for DECIMAL and NUMERIC data types. The field is undefined for all other data types.

SQL_DESC_SCHEMA_NAME [IRDs] This read-only SQLCHAR record field contains the schema name of the base table that contains the column. For many DBMS's, this is the owner name. If the data source does not support schemas (or owners) or the schema name cannot be determined, this variable contains an empty string.

SQL_DESC_SEARCHABLE [IRDs] This read-only SQLSMALLINT record field is set to one of the following values:

- SQL_PRED_NONE if the column cannot be used in a WHERE clause. (This is the same as the SQL_UNSEARCHABLE value defined in ODBC 2.0.)
- SQL_PRED_CHAR if the column can be used in a WHERE clause, but only with the LIKE predicate. (This is the same as the SQL_LIKE_ONLY value defined in ODBC 2.0.)
- SQL_PRED_BASIC if the column can be used in a WHERE clause with all the comparison operators except LIKE. (This is the same as the SQL_EXCEPT_LIKE value defined in ODBC 2.0.)
- SQL_PRED_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator.

SQL_DESC_TABLE_NAME [IRDs] This read-only SQLCHAR record field contains the name of the base table that contains this column.

SQL_DESC_TYPE [All] This SQLSMALLINT record field specifies the concise SQL or C data type for all data types.

Note: ODBC 3.0 defines the SQL_INTERVAL data type which is not supported by CLI. Any behavior associated with this data type is not present in CLI.

The values in the SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE fields are interdependent. Each time one of the fields is set, the other must also be set. SQL_DESC_TYPE can be set by a call to `SQLSetDescField()` or `SQLSetDescRec()`. SQL_DESC_CONCISE_TYPE can be set by a call to `SQLBindCol()` or `SQLBindParameter()`, or `SQLSetDescField()`.

If SQL_DESC_TYPE is set to a concise data type, the SQL_DESC_CONCISE_TYPE field is set to the same value, and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to 0.

When the SQL_DESC_TYPE field is set by a call to SQLSetDescField(), the following fields are set to the following default values. The values of the remaining fields of the same record are undefined:

Table 170. Default values

SQL_DESC_TYPE	Other fields Implicitly Set
SQL_CHAR, SQL_VARCHAR	SQL_DESC_LENGTH is set to 1. SQL_DESC_PRECISION is set to 0.
SQL_DECIMAL, SQL_NUMERIC	SQL_DESC_SCALE is set to 0. SQL_DESC_PRECISION is set to the precision for the data type.
SQL_FLOAT	SQL_DESC_PRECISION is set to the default precision for SQL_FLOAT.
SQL_DATETIME	SQL_DESC_CONCISE_TYPE, SQL_DESC_DATETIME_INTERVAL_CODE, or both may be set implicitly to indicate a DATE SQL or C type.
SQL_INTERVAL	This data type is not supported by CLI.

When an application calls SQLSetDescField() to set fields of a descriptor, rather than calling SQLSetDescRec(), the application must first declare the data type. If the values implicitly set are unacceptable, the application can then call SQLSetDescField() to set the unacceptable value explicitly.

SQL_DESC_TYPE_NAME [Implementation descriptors] This read-only SQLCHAR record field contains the data-source-dependent type name (for example, CHAR, VARCHAR, and so on). If the data type name is unknown, this variable contains an empty string.

SQL_DESC_UNNAMED [Implementation descriptors] This SQLSMALLINT record field in a row descriptor is set to either SQL_NAMED or SQL_UNNAMED. If the NAME field contains a column alias, or if the column alias does not apply, the UNNAMED field is set to SQL_NAMED. If there is no column name or a column alias, the UNNAMED field is set to SQL_UNNAMED.

An application can set the SQL_DESC_UNNAMED field of an IPD to SQL_UNNAMED. SQLSTATE HY091 (Invalid descriptor field identifier) is returned if an application attempts to set the SQL_DESC_UNNAMED field of an IPD to SQL_NAMED. The SQL_DESC_UNNAMED field of an IRD is read-only; SQLSTATE HY091 (Invalid descriptor field identifier) will be returned if an application attempts to set it.

SQL_DESC_UNSIGNED [Implementation descriptors] This read-only SQLSMALLINT record field is set to SQL_TRUE if the column type is unsigned or non-numeric, or SQL_FALSE if the column type is signed.

SQL_DESC_UPDATABLE [IRDs] This read-only SQLSMALLINT record field is set to one of the following values:
- SQL_ATTR_READONLY if the result set column is read-only.
- SQL_ATTR_WRITE if the result set column is read-write.
- SQL_ATTR_READWRITE_UNKNOWN if it is not known whether the result set column is updatable or not.

SQL_DESC_UPDATABLE describes the updatability of the column in the result set, not the column in the base table. The updatability of the column in the base table

on which this result set column is based may be different than the value in this field. Whether a column is updatable can be based on the data type, user privileges, and the definition of the result set itself. If it is unclear whether a column is updatable, SQL_UPDT_READWRITE_UNKNOWN should be returned.

SQL_DESC_USER_DEFINED_TYPE_CODE [IRDs] This read-only SQLINTEGER returns information that describes the nature of a column's data type. Four values may be returned:

- SQL_TYPE_BASE: the column data type is a base data type, such as CHAR, DATE, or DOUBLE.
- SQL_TYPE_DISTINCT: the column data type is a distinct user-defined type.
- SQL_TYPE_REFERENCE: the column data type is a reference user-defined type.
- SQL_TYPE_STRUCTURED: the column data type is a structured user-defined type.

SQL_DESC_CARDINALITY [APD, IPD] This SQLLEN record field indicates the maximum cardinality of the array value for the specified parameter marker. The IPD value indicates the maximum cardinality that can be sent to the database for an array value. The APD value indicates the maximum cardinality for an array value which is how the application indicates the maximum amount of allocated storage for output parameter array values for a CALL statement.

SQL_DESC_CARDINALITY_PTR [APD] This SQLLEN * record field points to a variable that will contain the cardinality of a parameter when the statement is executed. For input parameter markers, this is how the application provides the actual cardinality of an array value. For output parameter markers, this is the location where CLI will indicate the cardinality of the returned array value. Note that for output parameters, this value indicates the cardinality of the array returned by the stored procedure - not necessarily the cardinality written to the application, since it's possible for SQL_DESC_CARDINALITY(APD) to be less than the actual cardinality returned from the procedure. This is a deferred field - it is not used at the time it is set, but is used at a later time by CLI.

Descriptor header and record field initialization values (CLI)

The following tables list the initialization of each field for each type of descriptor, with D indicating that the field is initialized with a default, and ND indicating that the field is initialized without a default. If a number is shown, the default value of the field is that number. The tables also indicate whether a field is read/write (R/W) or read-only (R).

The initialization of header fields is as follows:

Table 171. Initialization of header fields

Descriptor header field	Type	Readable and writable (R/W) or read-only (R)	Initialization value
SQL_DESC_ALLOC_TYPE	SQLSMALLINT	• ARD: R • APD: R • IRD: R • IPD: R	• ARD: SQL_DESC_ALLOC_AUTO for implicit or SQL_DESC_ALLOC_USER for explicit • APD: SQL_DESC_ALLOC_AUTO for implicit or SQL_DESC_ALLOC_USER for explicit • IRD: SQL_DESC_ALLOC_AUTO • IPD: SQL_DESC_ALLOC_AUTO
SQL_DESC_ARRAY_SIZE	SQLUINTEGER	• ARD: R/W • APD: R/W • IRD: Unused • IPD: Unused	• ARD: [a] • APD: [a] • IRD: Unused • IPD: Unused
SQL_DESC_ARRAY_STATUS_PTR	SQLUSMALLINT *	• ARD: R/W • APD: R/W • IRD: R/W • IPD: R/W	• ARD: Null ptr • APD: Null ptr • IRD: Null ptr • IPD: Null ptr
SQL_DESC_BIND_OFFSET_PTR	SQLINTEGER *	• ARD: R/W • APD: R/W • IRD: Unused • IPD: Unused	• ARD: Null ptr • APD: Null ptr • IRD: Unused • IPD: Unused
SQL_DESC_BIND_TYPE	SQLINTEGER	• ARD: R/W • APD: R/W • IRD: Unused • IPD: Unused	• ARD: SQL_BIND_BY_COLUMN • APD: SQL_BIND_BY_COLUMN • IRD: Unused • IPD: Unused
SQL_DESC_COUNT	SQLSMALLINT	• ARD: R/W • APD: R/W • IRD: R • IPD: R/W	• ARD: 0 • APD: 0 • IRD: D • IPD: 0
SQL_DESC_ROWS_ PROCESSED_PTR	SQLUINTEGER *	• ARD: Unused • APD: Unused • IRD: R/W • IPD: R/W	• ARD: Unused • APD: Unused • IRD: Null Ptr • IPD: Null Ptr

a These fields are defined only when the IPD is automatically populated by CLI. If the fields are not automatically populated then they are undefined. If an application attempts to set these fields, SQLSTATE HY091 (Invalid descriptor field identifier.) will be returned.

The initialization of record fields is as follows:

Table 172. Initialization of record fields

Descriptor record field	Type	Readable and writable (R/W) or read-only (R)	Initialization value
SQL_DESC_AUTO_UNIQUE_VALUE	SQLINTEGER	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused

Table 172. Initialization of record fields (continued)

Descriptor record field	Type	Readable and writable (R/W) or read-only (R)	Initialization value
SQL_DESC_BASE_COLUMN_NAME	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused
SQL_DESC_BASE_TABLE_NAME	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused
SQL_DESC_CASE_SENSITIVE	SQLINTEGER	• ARD: Unused • APD: Unused • IRD: R • IPD: R	• ARD: Unused • APD: Unused • IRD: D • IPD: D [a]
SQL_DESC_CATALOG_NAME	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused
SQL_DESC_CONCISE_TYPE	SQLSMALLINT	• ARD: R/W • APD: R/W • IRD: R • IPD: R/W	• ARD: SQL_C_DEFAULT • APD: SQL_C_DEFAULT • IRD: D • IPD: ND
SQL_DESC_DATA_PTR	SQLPOINTER	• ARD: R/W • APD: R/W • IRD: Unused • IPD: Unused	• ARD: Null ptr • APD: Null ptr • IRD: Unused • IPD: Unused [b]
SQL_DESC_DATETIME_ INTERVAL_CODE	SQLSMALLINT	• ARD: R/W • APD: R/W • IRD: R • IPD: R/W	• ARD: ND • APD: ND • IRD: D • IPD: ND
SQL_DESC_DATETIME_ INTERVAL_PRECISION	SQLINTEGER	• ARD: R/W • APD: R/W • IRD: R • IPD: R/W	• ARD: ND • APD: ND • IRD: D • IPD: ND
SQL_DESC_DISPLAY_SIZE	SQLINTEGER	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused
SQL_DESC_FIXED_PREC_SCALE	SQLSMALLINT	• ARD: Unused • APD: Unused • IRD: R • IPD: R	• ARD: Unused • APD: Unused • IRD: D • IPD: D [a]
SQL_DESC_INDICATOR_PTR	SQLINTEGER *	• ARD: R/W • APD: R/W • IRD: Unused • IPD: Unused	• ARD: Null ptr • APD: Null ptr • IRD: Unused • IPD: Unused
SQL_DESC_LABEL	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused

Table 172. Initialization of record fields (continued)

Descriptor record field	Type	Readable and writable (R/W) or read-only (R)	Initialization value
SQL_DESC_LENGTH	SQLUINTEGER	• ARD: R/W • APD: R/W • IRD: R • IPD: R/W	• ARD: ND • APD: ND • IRD: D • IPD: ND
SQL_DESC_LITERAL_PREFIX	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused
SQL_DESC_LITERAL_SUFFIX	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused
SQL_DESC_LOCAL_TYPE_NAME	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: R	• ARD: Unused • APD: Unused • IRD: D • IPD: D [a]
SQL_DESC_NAME	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: R/W	• ARD: ND • APD: ND • IRD: D • IPD: ND
SQL_DESC_NULLABLE	SQLSMALLINT	• ARD: Unused • APD: Unused • IRD: R • IPD: R	• ARD: ND • APD: ND • IRD: N • IPD: ND
SQL_DESC_NUM_PREC_RADIX	SQLINTEGER	• ARD: R/W • APD: R/W • IRD: R • IPD: R/W	• ARD: ND • APD: ND • IRD: D • IPD: ND
SQL_DESC_OCTET_LENGTH	SQLINTEGER	• ARD: R/W • APD: R/W • IRD: R • IPD: R/W	• ARD: ND • APD: ND • IRD: D • IPD: ND
SQL_DESC_OCTET_LENGTH_PTR	SQLINTEGER *	• ARD: R/W • APD: R/W • IRD: Unused • IPD: Unused	• ARD: Null ptr • APD: Null ptr • IRD: Unused • IPD: Unused
SQL_DESC_PARAMETER_TYPE	SQLSMALLINT	• ARD: Unused • APD: Unused • IPD: Unused • IRD: R/W	• ARD: Unused • APD: Unused • IPD: Unused • IRD: D=SQL_PARAM_INPUT
SQL_DESC_PRECISION	SQLSMALLINT	• ARD: R/W • APD: R/W • IRD: R • IPD: R/W	• ARD: ND • APD: ND • IRD: D • IPD: ND
SQL_DESC_SCALE	SQLSMALLINT	• ARD: R/W • APD: R/W • IRD: R • IPD: R/W	• ARD: ND • APD: ND • IRD: D • IPD: ND

Table 172. Initialization of record fields (continued)

Descriptor record field	Type	Readable and writable (R/W) or read-only (R)	Initialization value
SQL_DESC_SCHEMA_NAME	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused
SQL_DESC_SEARCHABLE	SQLSMALLINT	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused
SQL_DESC_TABLE_NAME	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused
SQL_DESC_TYPE	SQLSMALLINT	• ARD: R/W • APD: R/W • IRD: R • IPD: R/W	• ARD: SQL_C_DEFAULT • APD: SQL_C_DEFAULT • IRD: D • IPD: ND
SQL_DESC_TYPE_NAME	SQLCHAR *	• ARD: Unused • APD: Unused • IRD: R • IPD: R	• ARD: Unused • APD: Unused • IRD: D • IPD: D [a]
SQL_DESC_UNNAMED	SQLSMALLINT	• ARD: Unused • APD: Unused • IRD: R • IPD: R/W	• ARD: ND • APD: ND • IRD: D • IPD: ND
SQL_DESC_UNSIGNED	SQLSMALLINT	• ARD: Unused • APD: Unused • IRD: R • IPD: R	• ARD: Unused • APD: Unused • IRD: D • IPD: D [a]
SQL_DESC_UPDATABLE	SQLSMALLINT	• ARD: Unused • APD: Unused • IRD: R • IPD: Unused	• ARD: Unused • APD: Unused • IRD: D • IPD: Unused
SQL_DESC_CARDINALITY	SQLLEN	• ARD: Unused • APD: R/W • IRD: Unused • IPD: R/W	• ARD: Unused • APD: D • IRD: Unused • IPD: D
SQL_DESC_CARDINALITY_PTR	SQLLEN *	• ARD: Unused • APD: R/W • IRD: Unused • IPD: Unused	• ARD: Unused • APD: D • IRD: Unused • IPD: Unused

a These fields are defined only when the IPD is automatically populated by CLI. If the fields are not automatically populated then they are undefined. If an application attempts to set these fields, SQLSTATE HY091 (Invalid descriptor field identifier.) will be returned.

b The SQL_DESC_DATA_PTR field in the IPD can be set to force a consistency check. In a subsequent call to `SQLGetDescField()` or `SQLGetDescRec()`, CLI is not required to return the value that SQL_DESC_DATA_PTR was set to.

Chapter 6. Header and record fields for the DiagIdentifier argument (CLI)

Header fields

You can specify the following header fields for the *DiagIdentifier* argument. The only diagnostic header fields that you can define for a descriptor field are SQL_DIAG_NUMBER and SQL_DIAG_RETURNCODE.

Table 173. Header fields for the DiagIdentifier argument

Header field	Return type	Description
SQL_DIAG_CURSOR_ROW_COUNT	SQLINTEGER	The count of rows in the cursor. The semantics of the field depend upon the `SQLGetInfo()` information types, which indicate which row counts are available for each cursor type (in the SQL_CA2_CRC_EXACT and SQL_CA2_CRC_APPROXIMATE bits): • SQL_DYNAMIC_CURSOR_ATTRIBUTES2 • SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2 • SQL_KEYSET_CURSOR_ATTRIBUTES2 • SQL_STATIC_CURSOR_ATTRIBUTES2 A value is defined in this field only for statement handles and only after a call to the SQLExecute(), SQLExecDirect(), or SQLMoreResults() function. Calling the SQLGetDiagField() function with a *DiagIdentifier* argument value of SQL_DIAG_CURSOR_ROW_COUNT on a handle other than a statement handle returns SQL_ERROR.
SQL_DIAG_DYNAMIC_FUNCTION	CHAR *	A string that describes the SQL statement that the underlying function executed (for the values that CLI supports, see Dynamic function fields). A value is defined in this field only for statement handles, and only after a call to the SQLExecute(), SQLExecDirect(), or SQLMoreResults() function.

Table 173. Header fields for the DiagIdentifier argument (continued)

Header field	Return type	Description
SQL_DIAG_DYNAMIC_ FUNCTION_CODE	SQLINTEGER	A numeric code that describes the SQL statement that was executed by the underlying function (for the values that CLI supports, see Dynamic function fields). A value is defined in this field only for statement handles, and only after a call to the SQLExecute(), SQLExecDirect(), or SQLMoreResults() function. Calling the SQLGetDiagField() function with a *DiagIdentifier* argument value of SQL_DIAG_DYNAMIC_FUNCTION_CODE on a handle other than a statement handle returns SQL_ERROR.
SQL_DIAG_NETWORK_STATISTICS	SQL_NET_STATS **Note:** The SQL_NET_STATS structure is defined in the `sqlcli1.h` file.	A structure containing network statistics for a connection. The statistics include the following information: • Database processing time in microseconds • Network time (including database processing time) in microseconds • Number of bytes that are sent to the database server • Number of bytes that are received from the database server • Number of DRDA round trips This field is available starting in Version 9.7, Fix Pack 3. CLI accumulates statistics for a connection when the SQL_ATTR_NETWORK_STATISTICS connection attribute is enabled. After an application calls the SQLGetDiagField() function to retrieve the statistics, CLI resets the internal counters that the connection uses to accumulate the statistics. A value is defined in this field only for connection handles, SQL_HANDLE_DBC. • Calling the SQLGetDiagField() function with a *DiagIdentifier* argument of SQL_DIAG_NETWORK_STATISTICS with an SQL_HANDLE_ENV returns ERROR. • Calling the SQLGetDiagField() function with a *DiagIdentifier* argument of SQL_DIAG_NETWORK_STATISTICS and a SQL_HANDLE_STMT or SQL_HANDLE_DESC returns the diagnostic identifier field for the underlying connection handle associated with the specified statement or descriptor handle.

Table 173. Header fields for the DiagIdentifier argument (continued)

Header field	Return type	Description
SQL_DIAG_NUMBER	SQLINTEGER	The number of status records that are available for the specified handle.
SQL_DIAG_RELATIVE_ COST_ESTIMATE	SQLINTEGER	A relative cost estimate of the resources that are required to process a statement if the SQLPrepare() function is invoked and successful. If deferred prepare is enabled, this field has the value 0 until the statement is executed.
SQL_DIAG_RETURNCODE	RETCODE	The return code of the last executed function that is associated with the specified handle. If a function has not been called on the *Handle*, SQL_SUCCESS is returned in the SQL_DIAG_RETURNCODE field.
SQL_DIAG_ROW_COUNT	SQLINTEGER	The number of rows that are affected by an insert, delete, or update performed by the SQLExecute(), SQLExecDirect(), or SQLSetPos() function. The value of this field is defined after a cursor specification has been executed and only for statement handles. The data in this field is returned in the *RowCountPtr* argument of the SQLRowCount() function. The data in this field is reset after every function call, whereas the row count returned by theSQLRowCount() function remains the same until the statement state is reset to the prepared or allocated state.
SQL_DIAG_TOLERATED_ERROR		Beginning with WebSphere® Federated Server V9.1, you can specify an Error Tolerant Nested Table Expression (ETNTE). This allows you to specify errors that can be tolerated and is of particular use when one or more data sources are unavailable during a `UNION ALL` query. By tolerating the absence of one of the unions (because the source is offline) a result set can still be processed and `SQLFetch` can be called repeatedly until it returns **`SQL_NO_DATA_FOUND`**(100). If tolerate errors are encountered then a CLI application can discover this by doing one of two things: • Call the `SQLGetDiagField()` function with a **`DiagIdentifier`** argument value of SQL_DIAG_TOLERATED_ERROR. If a tolerated error was processed, TRUE is returned. Otherwise, FALSE is returned. • Call the `SQLGetDiagRec()` function. If a tolerated error was processed, the diagnostic record displays error SQL20383W. Otherwise, error SQL0100W is displayed.

Record fields

You can specify the following record fields for the *DiagIdentifier* argument.

Table 174. Record fields for the DiagIdentifier argument

Record field	Return type	Description
SQL_DIAG_CLASS_ORIGIN	CHAR *	A string that indicates the document that defines the class and subclass portions of the SQLSTATE value in a record. CLI always returns an empty string in the SQL_DIAG_CLASS_ORIGIN field.
SQL_DIAG_COLUMN_NUMBER	SQLINTEGER	A value that represents the column number in a result set if the value of the SQL_DIAG_ROW_NUMBER field is a valid row number in a rowset or set of parameters. Result set column numbers always start at 1; if the status record pertains to a bookmark column, the value of the field can be zero. The field has the value SQL_NO_COLUMN_NUMBER if the status record is not associated with a column number. If CLI cannot determine the column number that a record is associated with, this field has the value SQL_COLUMN_NUMBER_UNKNOWN. A value is defined in this field only for statement handles.
SQL_DIAG_CONNECTION_NAME	CHAR *	A string that indicates the name of the connection that a diagnostic record relates to. CLI always returns an empty string in the SQL_DIAG_CONNECTION_NAME field.
SQL_DIAG_ERRMC	CHAR *	A string containing one or more message tokens that are separated by X'FF'.
SQL_DIAG_MESSAGE_TEXT	CHAR *	An informational message about an error or warning.
SQL_DIAG_NATIVE	SQLINTEGER	A native error code that is specific to a driver or data source, if a code exists; otherwise, the driver returns 0.
SQL_DIAG_ROW_NUMBER	SQLINTEGER	The row number in the rowset, or the parameter number in the set of parameters, with which the status record is associated. This field has the value SQL_NO_ROW_NUMBER if the status record is not associated with a row number. If CLI cannot determine the row number that a record is associated with, this field has the value SQL_ROW_NUMBER_UNKNOWN. A value is defined in this field only for statement handles.

Table 174. Record fields for the DiagIdentifier argument (continued)

Record field	Return type	Description
SQL_DIAG_SERVER_NAME	CHAR *	A string that indicates the server name that a diagnostic record relates to. The string is the same as the value returned for a call to the SQLGetInfo() function with an *InfoType* argument value of SQL_DATA_SOURCE_NAME. For diagnostic data structures associated with an environment handle and for diagnostics that do not relate to any server, this field is a 0-length string.
SQL_DIAG_SQLSTATE	CHAR *	A 5-character SQLSTATE diagnostic code.
SQL_DIAG_SUBCLASS_ORIGIN	CHAR *	A string, with the same format and valid values as the SQL_DIAG_CLASS_ORIGIN field, that identifies the defining portion of the subclass portion of the SQLSTATE code.

Values of the dynamic function fields

The following table describes the values of SQL_DIAG_DYNAMIC_FUNCTION and SQL_DIAG_DYNAMIC_FUNCTION_CODE that apply to each type of SQL statement that is executed by a call to the SQLExecute() or SQLExecDirect() function. CLI uses the values in this table; ODBC specifies other values.

Table 175. Values of dynamic function fields

SQL statement executed	Value of SQL_DIAG_ DYNAMIC_FUNCTION	Value of SQL_DIAG_DYNAMIC_ FUNCTION_CODE
alter-table-statement	ALTER TABLE	SQL_DIAG_ALTER_TABLE
create-index-statement	CREATE INDEX	SQL_DIAG_CREATE_INDEX
create-table-statement	CREATE TABLE	SQL_DIAG_CREATE_TABLE
create-view-statement	CREATE VIEW	SQL_DIAG_CREATE_VIEW
cursor-specification	SELECT CURSOR	SQL_DIAG_SELECT_CURSOR
delete-statement-positioned	DYNAMIC DELETE CURSOR	SQL_DIAG_DYNAMIC_DELETE_ CURSOR
delete-statement-searched	DELETE WHERE	SQL_DIAG_DELETE_WHERE
drop-index-statement	DROP INDEX	SQL_DIAG_DROP_INDEX
drop-table-statement	DROP TABLE	SQL_DIAG_DROP_TABLE
drop-view-statement	DROP VIEW	SQL_DIAG_DROP_VIEW
grant-statement	GRANT	SQL_DIAG_GRANT
insert-statement	INSERT	SQL_DIAG_INSERT
ODBC-procedure-extension	CALL	SQL_DIAG_PROCEDURE_CALL
revoke-statement	REVOKE	SQL_DIAG_REVOKE
update-statement-positioned	DYNAMIC UPDATE CURSOR	SQL_DIAG_DYNAMIC_UPDATE_ CURSOR
update--statement-searched	UPDATE WHERE	SQL_DIAG_UPDATE_WHERE
merge-statement	MERGE	SQL_DIAG_MERGE

Table 175. Values of dynamic function fields (continued)

SQL statement executed	Value of SQL_DIAG_ DYNAMIC_FUNCTION	Value of SQL_DIAG_DYNAMIC_ FUNCTION_CODE
Unknown	empty string	SQL_DIAG_UNKNOWN_ STATEMENT

Chapter 7. CLI data type attributes

SQL symbolic and default data types for CLI applications

The table lists each of the SQL data types that are used by Call Level Interface (CLI) applications, with its corresponding symbolic name, and the default C symbolic name.

SQL data type
This column contains the SQL data types as they would display in an SQL CREATE statement. The SQL data types are dependent on the DBMS.

Symbolic SQL data type
This column contains SQL symbolic names that are defined (in `sqlcli.h`) as an integer value. Various functions use these values to identify the SQL data types that are listed in the first column.

Default C symbolic data type
This column contains C symbolic names, which are also defined as integer values. These values are used in various function arguments to identify the C data type. Various functions use the symbolic names, such as `SQLBindParameter()`, `SQLGetData()`, and `SQLBindCol()`, to indicate the C data types of the application variables. Instead of explicitly identifying C data types when calling these functions, you can specify SQL_C_DEFAULT instead, and CLI assumes a default C data type based on the SQL data type of the parameter or column as shown by the following table. For example, the default C data type of SQL_DECIMAL is SQL_C_CHAR.

Applications should not use the `SQL_C_DEFAULT` data type to define C data types because it is less efficient for CLI. Explicitly indicating the C data type in the application is preferred, because it yields better performance than using `SQL_C_DEFAULT`.

Table 176. SQL symbolic and default data types

SQL data type	Symbolic SQL data type	Default symbolic C data type
BIGINT	SQL_BIGINT	SQL_C_SBIGINT
BINARY	SQL_BINARY	SQL_C_BINARY
BLOB	SQL_BLOB	SQL_C_BINARY
BLOB LOCATOR [a]	SQL_BLOB_LOCATOR	SQL_C_BLOB_LOCATOR
BOOLEAN [d]	SQL_BOOLEAN	SQL_C_DEFAULT
CHAR	SQL_CHAR	SQL_C_CHAR
CHAR	SQL_TINYINT	SQL_C_TINYINT
CHAR FOR BIT DATA [b]	SQL_BINARY	SQL_C_BINARY
CHAR FOR BIT DATA	SQL_BIT	SQL_C_BINARY
CLOB	SQL_CLOB	SQL_C_CHAR
CLOB LOCATOR [a]	SQL_CLOB_LOCATOR	SQL_C_CLOB_LOCATOR
CURSOR [d]	SQL_CURSORHANDLE	SQL_C_DEFAULT
DATE	SQL_TYPE_DATE	SQL_C_TYPE_DATE
DBCLOB	SQL_DBCLOB	SQL_C_DBCHAR
DBCLOB LOCATOR [a]	SQL_DBCLOB_LOCATOR	SQL_C_DBCLOB_LOCATOR
DECIMAL	SQL_DECIMAL	SQL_C_CHAR
DECFLOAT(16)	SQL_DECFLOAT	SQL_C_CHAR

Table 176. SQL symbolic and default data types (continued)

SQL data type	Symbolic SQL data type	Default symbolic C data type
DECFLOAT(34)	SQL_DECFLOAT	SQL_C_CHAR
DOUBLE	SQL_DOUBLE	SQL_C_DOUBLE
FLOAT	SQL_FLOAT	SQL_C_DOUBLE
GRAPHIC	SQL_GRAPHIC	SQL_C_DBCHAR
INTEGER	SQL_INTEGER	SQL_C_LONG
LONG VARCHAR [b]	SQL_LONGVARCHAR	SQL_C_CHAR
LONG VARCHAR FOR BIT DATA [b]	SQL_LONGVARBINARY	SQL_C_BINARY
LONG VARGRAPHIC [b]	SQL_LONGVARGRAPHIC	SQL_C_DBCHAR
LONG VARGRAPHIC [b]	SQL_WLONGVARCHAR	SQL_C_DBCHAR
NUMERIC [c]	SQL_NUMERIC [c]	SQL_C_CHAR
REAL	SQL_REAL	SQL_C_FLOAT
ROW [d]	SQL_ROW	SQL_C_DEFAULT
SMALLINT	SQL_SMALLINT	SQL_C_SHORT
TIME	SQL_TYPE_TIME	SQL_C_TYPE_TIME
TIMESTAMP	SQL_TYPE_TIMESTAMP	SQL_C_TYPE_TIMESTAMP
TIMESTAMP WITH TIMEZONE	SQL_TYPE_TIMESTAMP_ WITH_ TIMEZONE	SQL_C_TYPE_TIMESTAMP_ EXT_TZ
VARBINARY	SQL_VARBINARY	SQL_C_BINARY
VARCHAR	SQL_VARCHAR	SQL_C_CHAR
VARCHAR FOR BIT DATA [b]	SQL_VARBINARY	SQL_C_BINARY
VARGRAPHIC	SQL_VARGRAPHIC	SQL_C_DBCHAR
VARGRAPHIC	SQL_WVARCHAR	SQL_C_DBCHAR
WCHAR	SQL_WCHAR	SQL_C_WCHAR
XML[e]	SQL_XML	SQL_C_BINARY

- **a** LOB locator types are not persistent SQL data types. Columns cannot be defined with a locator type as they are only used to describe parameter markers, or to represent a LOB value.
- **b** LONG data types and FOR BIT DATA data types should be replaced by an appropriate LOB type whenever possible.
- **c** NUMERIC is a synonym for DECIMAL on DB2 for z/OS (Version 9.1 and later), DB2 Server for VSE & VM and DB2 Database for Linux, UNIX, and Windows.
- **d** BOOLEAN, CURSOR, and ROW types are only supported to provide applications with correct DESCRIBE information for database table columns or procedure parameters. No bind-in or bind-out is supported for these types. These types are recognized by the following CLI/ODBC APIs only: `SQLExtendedDescribe()` and `SQLDescribeParam()`.
- **e** In DB2 Version 9.7 Fix Pack 5, the SQL_XML data type is supported for DB2 for i V7R1 servers or later releases.
- **f** Starting in DB2 Version 9.7 Fix Pack 6, SQL_BINARY and SQL_VARBINARY data types are supported for DB2 for i Version 6 Release 1 servers or later releases.

Note:

The data types DATE, DECIMAL, DECFLOAT(16), DECFLOAT(34), NUMERIC, TIME, and TIMESTAMP cannot be transferred to their default C buffer types without a conversion.

C data types for CLI applications

The following table lists the generic type definitions for each symbolic C type that is used in CLI applications.

C symbolic data type
This column contains C symbolic names, defined as integer values. These values are used in various function arguments to identify the C data type shown in the last column.

C type
This column contains C defined types, defined in `sqlcli.h` using a C `typedef` statement. The values in this column should be used to declare all CLI related variables and arguments, in order to make the application more portable. Refer to Table 179 on page 516 for a list of additional symbolic data types used for function arguments.

Base C type
This column is shown for reference only. All variables and arguments should be defined using the symbolic types in the previous column since the base C type is platform dependent. Some of the values are C structures that are described in Table 178 on page 514.

Table 177. C data types

C symbolic data type	C type	Base C type
SQL_C_BINARY	SQLCHAR	unsigned char
SQL_C_BINARYXML	SQLCHAR	unsigned char
SQL_C_BIT	SQLCHAR	unsigned char or char (Value 1 or 0)
SQL_C_BLOB_LOCATOR [a]	SQLINTEGER	32-bit integer
SQL_C_CLOB_LOCATOR [a]	SQLINTEGER	32-bit integer
SQL_C_CHAR	SQLCHAR	unsigned char
SQL_C_DBCHAR	SQLDBCHAR	wchar_t
SQL_C_DBCLOB_LOCATOR	SQLINTEGER	32-bit integer
SQL_C_DECIMAL64	SQLDECIMAL64	see Table 178 on page 514
SQL_C_DECIMAL128	SQLDECIMAL128	see Table 178 on page 514
SQL_C_DOUBLE	SQLDOUBLE	double
SQL_C_FLOAT	SQLREAL	float
SQL_C_LONG	SQLINTEGER	32-bit integer
SQL_C_NUMERIC [b]	SQL_NUMERIC_STRUCT	see Table 178 on page 514
SQL_C_SBIGINT	SQLBIGINT	64-bit integer
SQL_C_SHORT	SQLSMALLINT	16-bit integer
SQL_C_TINYINT	SQLSCHAR	signed char (Range -128 to 127)
SQL_C_TYPE_DATE	DATE_STRUCT	see Table 178 on page 514
SQL_C_TYPE_TIME	TIME_STRUCT	see Table 178 on page 514
SQL_C_TYPE_TIMESTAMP	TIMESTAMP_STRUCT	see Table 178 on page 514
SQL_C_TYPE_TIMESTAMP_EXT	TIMESTAMP_STRUCT_EXT	see Table 178 on page 514
SQL_C_TYPE_TIMESTAMP_EXT_TZ	TIMESTAMP_STRUCT_EXT_TZ	see Table 178 on page 514
SQL_C_UBIGINT	SQLUBIGINT	unsigned 64-bit integer
SQL_C_ULONG	SQLUINTEGER	unsigned 32-bit integer
SQL_C_USHORT	SQLUSMALLINT	unsigned 16-bit integer
SQL_C_UTINYINT	SQLUCHAR	unsigned char
SQL_C_WCHAR	SQLWCHAR	wchar_t

Table 177. C data types (continued)

C symbolic data type	C type	Base C type

- **a** LOB Locator Types.
- **b** Windows only.

Note: SQL file reference data types (used in embedded SQL) are not needed in CLI.
Note: You use the SQL_C_BINARYXML C data type with the binary XML data transmission format, which is supported by DB2 CLI starting in DB2 z/OS Version 9.7 Fixpack 1, and in DB2 for Linux, UNIX, and Windows Version 9.7 Fixpack 5. The DB2 server must also be at a level that supports the binary XML format.

Table 178. C structures

C type	Generic structure	Windows structure

DATE_STRUCT	`typedef struct DATE_STRUCT`
	`  {`
	`  SQLSMALLINT  year;`
	`  SQLUSMALLINT  month;`
	`  SQLUSMALLINT  day;`
	`  } DATE_STRUCT;`
		`typedef struct tagDATE_STRUCT`
		`  {`
		`  SWORD  year;`
		`  UWORD  month;`
		`  UWORD  day;`
		`  } DATE_STRUCT;`

TIME_STRUCT	`typedef struct TIME_STRUCT`
	`  {`
	`  SQLUSMALLINT  hour;`
	`  SQLUSMALLINT  minute;`
	`  SQLUSMALLINT  second;`
	`  } TIME_STRUCT;`
		`typedef struct tagTIME_STRUCT`
		`  {`
		`  UWORD  hour;`
		`  UWORD  minute;`
		`  UWORD  second;`
		`  } TIME_STRUCT;`

TIMESTAMP_STRUCT	`typedef struct TIMESTAMP_STRUCT`
	`  {`
	`  SQLUSMALLINT  year;`
	`  SQLUSMALLINT  month;`
	`  SQLUSMALLINT  day;`
	`  SQLUSMALLINT  hour;`
	`  SQLUSMALLINT  minute;`
	`  SQLUSMALLINT  second;`
	`  SQLINTEGER  fraction;`
	`  } TIMESTAMP_STRUCT;`
		`typedef struct tagTIMESTAMP_STRUCT`
		`  {`
		`  SWORD  year;`
		`  UWORD  month;`
		`  UWORD  day;`
		`  UWORD  hour;`
		`  UWORD  minute;`
		`  UWORD  second;`
		`  UDWORD  fraction;`
		`  } TIMESTAMP_STRUCT;`

TIMESTAMP_STRUCT_EXT	`typedef  struct TIMESTAMP_STRUCT_EXT`
	`  {`
	`  SQLSMALLINT  year;`
	`  SQLUSMALLINT  month;`
	`  SQLUSMALLINT  day;`
	`  SQLUSMALLINT  hour;`
	`  SQLUSMALLINT  minute;`
	`  SQLUSMALLINT  second;`
	`  SQLUINTEGER fraction;`
	`/* 1-9 digits */`
	`  SQLUINTEGER  fraction2;`
	`/* 10-12 digits */  }`
	`TIMESTAMP_STRUCT_EXT;`
		(No Windows structure. Only a generic structure.)

Table 178. C structures (continued)

C type	Generic structure	Windows structure

TIMESTAMP_STRUCT_EXT_TZ	typedef struct TIMESTAMP_STRUCT
	{
	SQLSMALLINT year;
	SQLUSMALLINT month;
	SQLUSMALLINT day;
	SQLUSMALLINT hour;
	SQLUSMALLINT minute;
	SQLUSMALLINT second;
	SQLUINTEGER fraction;
	SQLUINTEGER fraction2;
	SQLSMALLINT timezone_hour;
	/*-12 to 14*/
	SQLUSMALLINT timezone_minute;
	/*-59 to 59*/
	}
	TIMESTAMP_STRUCT_EXT_TZ;
		(No Windows structure. Only a generic structure.)

SQLDECIMAL64	typedef struct tagSQLDECIMAL64
	{
	union {
	SQLDOUBLE dummy;
	SQLCHAR dec64
	[SQL_DECFLOAT16_
	COEFFICIENT_LEN];
	} udec64;
	} SQLDECIMAL64;
		(No Windows structure. Only a generic structure.)

SQLDECIMAL128	typedef struct tagSQLDECIMAL128
	{
	union {
	SQLDOUBLE dummy;
	SQLCHAR dec128
	[SQL_DECFLOAT34_
	COEFFICIENT_LEN];
	} udec128;
	} SQLDECIMAL128;
		(No Windows structure. Only a generic structure.)

SQL_NUMERIC_STRUCT	(No generic structure. Only a Windows structure.)	typedef struct tagSQL_NUMERIC_STRUCT { SQLCHAR precision; SQLCHAR scale; SQLCHAR sign; [a] SQLCHAR val[SQL_MAX_NUMERIC_LEN];[b] [c] } SQL_NUMERIC_STRUCT;

Refer to Table 179 on page 516 for more information about the SQLUSMALLINT C data type.

- **a** Sign field: 1 = positive, 2 = negative
- **b** A number is stored in the val field of the SQL_NUMERIC_STRUCT structure as a scaled integer, in little endian mode (the leftmost byte being the least-significant byte). For example, the number 10.001 base 10, with a scale of 4, is scaled to an integer of 100010. Because this is 186AA in hexadecimal format, the value in SQL_NUMERIC_STRUCT would be "AA 86 01 00 00 ... 00", with the number of bytes defined by the SQL_MAX_NUMERIC_LEN #define.
- **c** The precision and scale fields of the SQL_C_NUMERIC data type are never used for input from an application, only for output from the driver to the application. When the driver writes a numeric value into the SQL_NUMERIC_STRUCT, it will use its own default as the value for the precision field, and it will use the value in the SQL_DESC_SCALE field of the application descriptor (which defaults to 0) for the scale field. An application can provide its own values for precision and scale by setting the SQL_DESC_PRECISION and SQL_DESC_SCALE fields of the application descriptor.

C data types for CLI applications

As well as the data types that map to SQL data types, there are also C symbolic types used for other function arguments such as pointers and handles. Both the generic and ODBC data types are shown in the following table.

Note: There are two kinds of drivers that ship with the product: the CLI driver, and the 64-bit ODBC driver. The 64-bit ODBC Driver handles the differences with type definitions between various ODBC Managers.

Table 179. C Data types and base C data types

Defined C type	Base C type	Typical usage
SQLPOINTER	void *	Pointer to storage for data and parameters.
SQLHANDLE	1. 1. void * 2. 32-bit integer	Handle used to reference all 4 types of handle information. 1. 64-bit value for Windows 64-bit ODBC Driver and UNIX 64-bit ODBC Driver 2. 32-bit value for all 32-bit platforms and 64-bit CLI Drivers
SQLHENV	1. 1. void * 2. 32-bit integer	Handle referencing environment information. 1. 64-bit value for Windows 64-bit ODBC Driver and UNIX 64-bit ODBC Driver 2. 32-bit value for all 32-bit platforms and 64-bit CLI Drivers
SQLHDBC	1. 1. void * 2. 32-bit integer	Handle referencing database connection information. 1. 64-bit value for Windows 64-bit ODBC Driver and UNIX 64-bit ODBC Driver 2. 32-bit value for all 32-bit platforms and 64-bit CLI Drivers
SQLHSTMT	1. 1. void * 2. 32-bit integer	Handle referencing statement information. 1. 64-bit value for Windows 64-bit ODBC Driver and UNIX 64-bit ODBC Driver 2. 32-bit value for all 32-bit platforms and 64-bit CLI Drivers
SQLUSMALLINT	unsigned 16-bit integer	Function input argument for unsigned short integer values.
SQLUINTEGER	unsigned 32-bit integer	Function input argument for unsigned long integer values.
SQLRETURN	16-bit integer	Return code from CLI functions.
SQLULEN	1. unsigned 64-bit integer 2. unsigned 32-bit integer	1. Function input or output argument for unsigned 64-bit integer values (64-bit ODBC driver). 2. Function input or output argument for unsigned 32-bit integer values (all other drivers).
SQLLEN	1. 64-bit integer 2. 32-bit integer	1. Function input or output argument for 64-bit integer values (64-bit ODBC driver). 2. Function input or output argument for 32-bit integer values (all other drivers).

Table 179. C Data types and base C data types (continued)

Defined C type	Base C type	Typical usage
SQLSETPOSIROW	1. 64-bit integer 2. 16-bit integer	1. Function input or output argument for 64-bit integer values (Windows 64-bit ODBC driver). 2. Function input or output argument for 16-bit integer values (all other drivers).

Data conversions supported in CLI

The table lists data type conversions supported by CLI.

The first column contains the data type of the SQL data type. The remaining columns represent the C data types. If the C data type columns contain:

D The conversion is supported and this is the default conversion for the SQL data type.

X All IBM DBMSs support the conversion.

blank No IBM DBMS supports the conversion.

See the tables of data type attributes (precision, scale, length, and display) for more information about the data type formats.

Table 180. Supported data conversions

SQL data type	SQL_C_CHAR	SQL_C_WCHAR	SQL_C_BINARYXML	SQL_C_DBCHAR	SQL_C_BIT	SQL_C_TINYINT	SQL_C_STINYINT	SQL_C_UTINYINT	SQL_C_SHORT	SQL_C_SSHORT	SQL_C_USHORT	SQL_C_LONG	SQL_C_SLONG	SQL_C_ULONG	SQL_C_SBIGINT	SQL_C_UBIGINT	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_NUMERIC	SQL_C_DECIMAL_IBM	SQL_C_DECIMAL64	SQL_C_DECIMAL128	SQL_C_TYPE_DATE	SQL_C_TYPE_TIME	SQL_C_TYPE_TIMESTAMP	SQL_C_DATALINK	SQL_C_BINARY	SQL_C_CLOB_LOCATOR	SQL_C_BLOB_LOCATOR	SQL_C_DBCLOB_LOCATOR
SQL_CHAR	D	X			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X		X			
SQL_BINARYXML			D																											
SQL_WCHAR[4]	D	X			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X		X			
SQL_ VARCHAR	D	X			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X		X			
SQL_ WVARCHAR[4]	D	X			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X		X			
SQL_ LONG VARCHAR	D	X			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X		X			
SQL_ BINARY	X	X																									D			
SQL_ VARBINARY	X	X																									D			

Table 180. Supported data conversions (continued)

SQL data type	SQL_C_CHAR	SQL_C_WCHAR	SQL_C_BINARYXML	SQL_C_DBCHAR	SQL_C_BIT	SQL_C_TINYINT	SQL_C_STINYINT	SQL_C_UTINYINT	SQL_C_SHORT	SQL_C_SSHORT	SQL_C_USHORT	SQL_C_LONG	SQL_C_SLONG	SQL_C_ULONG	SQL_C_SBIGINT	SQL_C_UBIGINT	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_NUMERIC	SQL_C_DECIMAL_IBM	SQL_C_DECIMAL64	SQL_C_DECIMAL128	SQL_C_TYPE_DATE	SQL_C_TYPE_TIME	SQL_C_TYPE_TIMESTAMP	SQL_C_DATALINK	SQL_C_BINARY	SQL_C_CLOB_LOCATOR	SQL_C_BLOB_LOCATOR	SQL_C_DBCLOB_LOCATOR
SQL_LONG VARBINARY	X	X																									D			
SQL_GRAPHIC	X	X		D					X		X	X		X	X	X	X	X									X			
SQL_VARGRAPHIC	X	X		D					X		X	X		X	X	X	X	X									X			
SQL_LONG VARGRAPHIC	X	X		D					X		X	X		X	X	X	X	X									X			
SQL_CLOB	D	X																									X	X		
SQL_BLOB	X	X																									D		X	
SQL_DBCLOB	X	X		D																							X			X
SQL_CLOB_LOCATOR																												D		
SQL_BLOB_LOCATOR																													D	
SQL_DBCLOB_LOCATOR																														D
SQL_NUMERIC	D	X		X[5]	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X					X			
SQL_DECIMAL	D	X		X[5]	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X					X			
SQL_DECFLOAT	D	X		X[5]	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X								
SQL_INTEGER	X	X		X[5]	X	X	X	X	X	X	X	D[3]	D[3]	X	X	X	X	X	X		X	X					X			
SQL_SMALLINT	X	X		X[5]	X	X	X	X	D[2]	D[2]	X	X	X	X	X	X	X	X	X		X	X					X			
SQL_FLOAT	X	X		X[5]	X	X	X	X	X	X	X	X	X	X	X	X	X	D	X		X	X					X			
SQL_DOUBLE	X	X		X[5]	X	X	X	X	X	X	X	X	X	X	X	X	X	D	X		X	X					X			
SQL_REAL	X	X		X[5]	X	X	X	X	X	X	X	X	X	X	X	X	D	X	X		X	X					X			
SQL_BIGINT	X	X		X[5]	X	X	X	X	X	X	X	X	X	X	D	X	X	X	X		X	X					X			

Table 180. Supported data conversions (continued)

SQL data type	SQL_C_CHAR	SQL_C_WCHAR	SQL_C_BINARYXML	SQL_C_DBCHAR	SQL_BIT	SQL_C_TINYINT	SQL_C_STINYINT	SQL_C_UTINYINT	SQL_C_SHORT	SQL_C_SSHORT	SQL_C_USHORT	SQL_C_LONG	SQL_C_SLONG	SQL_C_ULONG	SQL_C_SBIGINT	SQL_C_UBIGINT	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_NUMERIC	SQL_C_DECIMAL_IBM	SQL_C_DECIMAL64	SQL_C_DECIMAL128	SQL_C_TYPE_DATE	SQL_C_TYPE_TIME	SQL_C_TYPE_TIMESTAMP	SQL_C_DATALINK	SQL_C_BINARY	SQL_C_CLOB_LOCATOR	SQL_C_BLOB_LOCATOR	SQL_C_DBCLOB_LOCATOR
SQL_TINYINT	X	X			X	D[1]	D[1]	X	X	X	X	X	X	X	X	X	X	X	X		X	X					X			
SQL_BIT	X	X																									D			
SQL_ TYPE_DATE	X	X																					D		X					
SQL_ TYPE_TIME	X	X																						D	X					
SQL_ TYPE_ TIMESTAMP	X	X																					X	X	D					
SQL_XML	X	X	D	X																							D			

Note:

1. The C type SQL_C_TINYINT is treated by CLI as the type SQL_C_STINYINT so either type can be considered to be the default.
2. The C type SQL_C_SHORT is treated by CLI as the type SQL_C_SSHORT so either type can be considered to be the default.
3. The C type SQL_C_LONG is treated by CLI as the type SQL_C_SLONG so either type can be considered to be the default.
4. This type is not supported in the DB2 database but other supported types can be returned to the client as this type because of the setting of a configuration keyword, such as MAPCharToWChar.
5. This type conversion is restricted to Unicode.

The SQL_C_BINARYXML C data type is available for use with the binary XML data in DB2 for z/OS and DB2 for Linux, UNIX, and Windows. The DB2 server must also be at a level that supports the binary XML data type. The SQL_XML data type is not supported for use with an Informix data server.

- Starting in DB2 Version 9.7 Fix Pack 5, the SQL_XML data type is supported for DB2 for i Version 7 Release 1 servers or later releases.
- Starting in DB2 Version 9.7 Fix Pack 6, SQL_BINARY and SQL_VARBINARY data types are supported for DB2 for i Version 6 Release 1 servers or later releases.

The SQL_XML data type is supported for DB2 for i V7R1 servers or later releases

SQL to C data conversion in CLI

For a given SQL data type:

- the first column of the table lists the legal input values of the *fCType* argument in `SQLBindCol()` and `SQLGetData()`.
- the second column lists the outcomes of a test, often using the *cbValueMax* argument specified in `SQLBindCol()` or `SQLGetData()`, which the driver performs to determine if it can convert the data.
- the third and fourth columns list the values (for each outcome) of the *rgbValue* and *pcbValue* arguments specified in the `SQLBindCol()` or `SQLGetData()` after the driver has attempted to convert the data.
- the last column lists the SQLSTATE returned for each outcome by `SQLFetch()`, `SQLExtendedFetch()`, `SQLGetData()` or `SQLGetSubString()`.

The tables list the conversions defined by ODBC to be valid for a given SQL data type.

If the *fCType* argument in `SQLBindCol()` or `SQLGetData()` contains a value not shown in the table for a given SQL data type, `SQLFetch()`, or `SQLGetData()` returns the SQLSTATE 07006 (Restricted data type attribute violation).

If the *fCType* argument contains a value shown in the table but which specifies a conversion not supported by the driver, `SQLFetch()`, or `SQLGetData()` returns SQLSTATE HYC00 (Driver not capable).

Though it is not shown in the tables, the *pcbValue* argument contains SQL_NULL_DATA when the SQL data value is NULL. For an explanation of the use of *pcbValue* when multiple calls are made to retrieve data, see `SQLGetData()`.

When SQL data is converted to character C data, the character count returned in *pcbValue* does not include the null termination byte. If *rgbValue* is a null pointer, `SQLBindCol()` or `SQLGetData()` returns SQLSTATE HY009 (Invalid argument value).

In the following tables:

Length of data
: the total length of the data after it has been converted to the specified C data type (excluding the null termination byte if the data was converted to a string). This is true even if data is truncated before it is returned to the application.

Significant digits
: the minus sign (if needed) and the digits to the left of the decimal point.

Display size
: the total number of bytes needed to display data in the character format.

Converting character SQL data to C data

The character SQL data types are:

- SQL_CHAR
- SQL_VARCHAR
- SQL_LONGVARCHAR
- SQL_CLOB

Table 181. Converting character SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Length of data < cbValueMax	Data	Length of data	00000
	Length of data >= cbValueMax	Truncated data	Length of data	01004
SQL_C_BINARY	Length of data <= cbValueMax	Data	Length of data	00000
	Length of data > cbValueMax	Truncated data	Length of data	01004
SQL_C_SHORT SQL_C_LONG SQL_C_FLOAT SQL_C_FLOAT SQL_C_TINYINT SQL_C_BIT SQL_C_UBIGINT SQL_C_SBIGINT SQL_C_NUMERIC [c]	Data converted without truncation [a]	Data	Size of the C data type	00000
	Data converted with truncation, but without loss of significant digits [a]	Data	Size of the C data type	01004
	Conversion of data would result in loss of significant digits[a]	Untouched	Size of the C data type	22003
	Data is not a number [a]	Untouched	Size of the C data type	22005
SQL_C_TYPE_DATE	Data value is a valid date [a]	Data	6 [b]	00000
	Data value is not a valid date [a]	Untouched	6 [b]	22007
SQL_C_TYPE_TIME	Data value is a valid time [a]	Data	6 [b]	00000
	Data value is not a valid time [a]	Untouched	6 [b]	22007
SQL_C_TYPE_TIMESTAMP	Data value is a valid timestamp [a]	Data	16 [b]	00000
	Data value is not a valid timestamp [a]	Untouched	16 [b]	22007
SQL_C_TIMESTAMP_EXT	Data value is a valid timestamp [a]	Data	20	00000
	Data value is not a valid timestamp [a]	Untouched	20	22007

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] SQL_C_NUMERIC is only supported on Windows platforms.

SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting graphic SQL data to C data

The graphic SQL data types are:

- SQL_GRAPHIC
- SQL_VARGRAPHIC
- SQL_LONGVARGRAPHIC
- SQL_DBCLOB

Table 182. Converting GRAPHIC SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Number of double byte characters * 2 <= cbValueMax	Data	Length of data(octects)	00000
	Number of double byte characters * 2 > cbValueMax	Truncated data, to the nearest even byte that is less than *cbValueMax*.	Length of data(octects)	01004
SQL_C_DBCHAR	Number of double byte characters * 2 < cbValueMax	Data	Length of data(octects)	00000
	Number of double byte characters * 2 >= cbValueMax	Truncated data, to the nearest even byte that is less than *cbValueMax*.	Length of data(octects)	01004

Note: SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

When converting to floating point values, SQLSTATE 22003 will not be returned if non-significant digits of the resulting value are lost.

Converting numeric SQL data to C data

The numeric SQL data types are:

- SQL_DECIMAL
- SQL_NUMERIC
- SQL_SMALLINT
- SQL_INTEGER
- SQL_BIGINT
- SQL_REAL
- SQL_DECFLOAT
- SQL_FLOAT
- SQL_DOUBLE

Table 183. Converting numeric SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Length of data	00000
	Number of significant digits < cbValueMax	Truncated data	Length of data	01004
	Number of significant digits >= cbValueMax	Untouched	Length of data	22003
SQL_C_DBCHAR SQL_C_WCHAR	Display size * 2 < cbValueMax	Data	Length of data	00000
	Number of significant digits * 2 < cbValueMax	Truncated Data	Length of Data	01004
	Number of significant digits * 2 >= cbValueMax	Untouched	Length of Data	22003

Table 183. Converting numeric SQL data to C data (continued)

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_SHORT SQL_C_LONG SQL_C_FLOAT SQL_C_DOUBLE SQL_C_TINYINT SQL_C_BIT SQL_C_UBIGINT SQL_C_SBIGINT SQL_C_NUMERIC [b]	Data converted without truncation [a]	Data	Size of the C data type	00000
	Data converted with truncation, but without loss of significant digits [a]	Truncated data	Size of the C data type	01004
	Conversion of data would result in loss of significant digits [a]	Untouched	Size of the C data type	22003

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] SQL_C_NUMERIC is only supported on Windows platforms.

SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting binary SQL data to C data

The binary SQL data types are:

- SQL_BINARY
- SQL_VARBINARY
- SQL_LONGVARBINARY
- SQL_BLOB

Table 184. Converting binary SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	(Length of data) < cbValueMax	Data	Length of data	N/A
	(Length of data) >= cbValueMax	Truncated data	Length of data	01004
SQL_C_BINARY	Length of data <= cbValueMax	Data	Length of data	N/A
	Length of data > cbValueMax	Truncated data	Length of data	01004

Note: Starting in DB2 Version 9.7 Fix Pack 6, SQL_BINARY and SQL_VARBINARY data types are supported for DB2 for i Version 6 Release 1 servers or later releases.

Converting XML SQL data to C data

The XML SQL data type is:

SQL_XML

Table 185. Converting XML SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Length of data < cbValueMax	Data	Length of data	00000
	Length of data >= cbValueMax	Truncated data	Length of data	01004

Table 185. Converting XML SQL data to C data (continued)

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_BINARY	Length of data <= cbValueMax	Data	Length of data	00000
	Length of data > cbValueMax	Truncated data	Length of data	01004
SQL_C_BINARYXML	Length of data <= cbValueMax	Data	Length of data	00000
	Length of data > cbValueMax	Truncated data	Length of data	01004
SQL_C_DBCHAR	Number of double-byte characters * 2 < cbValueMax	Data	Length of data	00000
	Number of double-byte characters * 2 >= cbValueMax	Truncated data, to the nearest even byte that is less than cbValueMax	Length of data	01004
SQL_C_WCHAR	Number of double-byte characters * 2 < cbValueMax	Data	Length of data	00000
	Number of double-byte characters * 2 >= cbValueMax	Truncated data, to the nearest even byte that is less than cbValueMax	Length of data	01004

Note:

1. SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.
2. Length of data includes any XML declaration in the target encoding.
3. The SQL_XML data type is not supported for use with an Informix data server.

Converting date SQL data to C data

The date SQL data type is:

- SQL_TYPE_DATE

Table 186. Converting date SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 11	Data	10	00000
	cbValueMax < 11	Untouched	10	22003
SQL_C_TYPE_DATE	None [a]	Data	6 [b]	00000
SQL_C_TYPE_TIMESTAMP	None [a]	Data [c]	16 [b]	00000
SQL_C_TIMESTAMP_EXT	None [a]	Data [c]	20	00000

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] The time fields of the TIMESTAMP_STRUCT or TIMESTAMP_STRUCT_EXT structure are set to zero.

SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

When the date SQL data type is converted to the character C data type, the resulting string is in the "yyyy-mm-dd" format.

Converting Time SQL Data to C Data

The time SQL data type is:

- SQL_TYPE_TIME

Table 187. Converting time SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 9	Data	8	00000
	cbValueMax < 9	Untouched	8	22003
SQL_C_TYPE_TIME	None [a]	Data	6 [b]	00000
SQL_C_TYPE_TIMESTAMP	None [a]	Data [c]	16 [b]	00000
SQL_C_TIMESTAMP_EXT	None [a]	Data [c]	20	00000

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] The date fields of the TIMESTAMP_STRUCT or TIMESTAMP_STRUCT_EXT structure are set to the current system date of the machine that the application is running, and the time fraction is set to zero.

SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

When the time SQL data type is converted to the character C data type, the resulting string is in the "hh:mm:ss" format.

Converting timestamp SQL data to C data

The timestamp SQL data type is:

- SQL_TYPE_TIMESTAMP

Table 188. Converting timestamp SQL data to C data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Length of data	00000
	19 <= cbValueMax <= Display size	Truncated Data [b]	Length of data	01004
	cbValueMax < 19	Untouched	Length of data	22003
SQL_C_TYPE_DATE	None [a]	Truncated data [c]	6 [e]	01004
SQL_C_TYPE_TIME	None [a]	Truncated data [d]	6 [e]	01004
SQL_C_TYPE_TIMESTAMP	None [a]	Data	16 [e]	00000
SQL_C_TIMESTAMP_EXT	None [a]	Data	20	00000

Table 188. Converting timestamp SQL data to C data (continued)

fCType	Test	rgbValue	pcbValue	SQLSTATE

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] The fractional seconds of the timestamp are truncated.

[c] The time portion of the timestamp is deleted.

[d] The date portion of the timestamp is deleted.

[e] This is the size of the corresponding C data type.

SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

When the timestamp SQL data type is converted to the character C data type, the resulting string is in the "yyyy-mm-dd hh:mm:ss.fffffffffff" format where fractional second digits range from 0 to 12 (regardless of the precision of the timestamp SQL data type). If an application requires the ISO format, set the CLI/ODBC configuration keyword PATCH2=33.

Converting timestamp(p) with timezone SQL data to C data

The timestamp SQL data type is:

- SQL_TYPE_TIMESTAMP_WITH_TIMEZONE

Table 189. Converting timestamp(p) with timezone SQL data to C data

fCType	SQL Type	Test/Result	SQLSTATE
SQL_C_TYPE_TIMESTAMP_ EXT_TZ	SQL_CHAR	Data value is a valid timestamp with timezone	N/A
		Data value is not a valid timestamp with timezone	22007
SQL_C_TYPE_DATE	SQL_TYPE_TIMESTAMP_ WITH_TIMEZONE	Truncated data	01S07
SQL_C_TYPE_TIME	SQL_TYPE_TIMESTAMP_ WITH_TIMEZONE	Truncated data	01S07
SQL_C_TYPE_TIMESTAMP	SQL_TYPE_TIMESTAMP_ WITH_TIMEZONE	Truncated data	01S07

SQL to C data conversion examples

Table 190. SQL to C data conversion examples

SQL data type	SQL data value	C data type	cbValue max	rgbValue	SQL STATE
SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef\0 [a]	00000
SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0 [a]	00000
SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	4	---	22003
SQL_DECIMAL	1234.56	SQL_C_FLOAT	ignored	1234.56	00000
SQL_DECIMAL	1234.56	SQL_C_SHORT	ignored	1234	01004

Table 190. SQL to C data conversion examples (continued)

SQL data type	SQL data value	C data type	cbValue max	rgbValue	SQL STATE
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0 [a]	00000
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	10	---	22003
SQL_TYPE_DATE	1992-12-31	SQL_C_TYPE_ TIMESTAMP	ignored	1992,12,31, 0,0,0,0 [b]	00000
SQL_TYPE_ TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12\0 [a]	00000
SQL_TYPE_ TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1\0 [a]	01004
SQL_TYPE_ TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	---	22003

Note:

[a] "\0" represents a null termination character.

[b] The numbers in this list are the numbers stored in the fields of the TIMESTAMP_STRUCT or TIMESTAMP_STRUCT_EXT structure.

SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

C to SQL data conversion in CLI

For a given C data type:

- the first column of the table lists the legal input values of the *fSqlType* argument in `SQLBindParameter()` or `SQLSetParam()`.
- the second column lists the outcomes of a test, often using the length of the parameter data as specified in the *pcbValue* argument in `SQLBindParameter()` or `SQLSetParam()`, which the driver performs to determine if it can convert the data.
- the third column lists the SQLSTATE returned for each outcome by `SQLExecDirect()` or `SQLExecute()`.

The tables list the conversions defined by ODBC to be valid for a given SQL data type.

If the *fSqlType* argument in `SQLBindParameter()` or `SQLSetParam()` contains a value not shown in the table for a given C data type, SQLSTATE 07006 is returned (Restricted data type attribute violation).

If the *fSqlType* argument contains a value shown in the table but which specifies a conversion not supported by the driver, `SQLBindParameter()` or `SQLSetParam()` returns SQLSTATE HYC00 (Driver not capable).

If the *rgbValue* and *pcbValue* arguments specified in `SQLBindParameter()` or `SQLSetParam()` are both null pointers, that function returns SQLSTATE HY009 (Invalid argument value).

Note:

- The SQL_XML data type is not supported for use with an Informix data server.
- Starting in DB2 Version 9.7 Fix Pack 6, SQL_BINARY and SQL_VARBINARY data types are supported for DB2 for i Version 6 Release 1 servers or later releases.

Length of data
the total length of the data after it has been converted to the specified SQL data type (excluding the null termination byte if the data was converted to a string). This is true even if data is truncated before it is sent to the data source.

Column length
the maximum number of bytes returned to the application when data is transferred to its default C data type. For character data, the length does not include the null termination byte.

Display size
the maximum number of bytes needed to display data in character form.

Significant digits
the minus sign (if needed) and the digits to the left of the decimal point.

Converting character C data to SQL data

The character C data type is:

- SQL_C_CHAR

Table 191. Converting character C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Length of data <= Column length	N/A
	Length of data > Column length	22001
SQL_DECIMAL SQL_NUMERIC SQL_SMALLINT SQL_INTEGER SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DOUBLE	Data converted without truncation	N/A
	Data converted with truncation, but without loss of significant digits	22001
	Conversion of data would result in loss of significant digits	22003
	Data value is not a numeric value	22005
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	(Length of data) < Column length	N/A
	(Length of data) >= Column length	22001
	Data value is not a hexadecimal value	22005
SQL_TYPE_DATE	Data value is a valid date	N/A
	Data value is not a valid date	22007
	Data value is a valid timestamp	22008
	Data value is a valid timestamp and the connection attribute SQL_ATTR_REPORT_TIMESTAMP_TRUNC_AS_WARN is set to 1	01S07 (Fractional truncation warning)
SQL_TYPE_TIME	Data value is a valid time	N/A
	Data value is not a valid time	22007
	Data value is a valid timestamp	22008
	Data value is a valid timestamp and the connection attribute SQL_ATTR_REPORT_TIMESTAMP_TRUNC_AS_WARN is set to 1	01S07 (Fractional truncation warning)
SQL_TYPE_TIMESTAMP	Data value is a valid timestamp	N/A
	Data value is not a valid timestamp	22007
	Data value is a valid date	N/A
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC SQL_DBCLOB	Length of data / 2 <= Column length	N/A
	Length of data / 2 < Column length	22001
SQL_XML	Data can be implicitly parsed	(several SQLSTATES can be returned)

Converting numeric C data to SQL data

The numeric C data types are:

- SQL_C_SHORT

- SQL_C_LONG
- SQL_C_FLOAT
- SQL_C_DOUBLE
- SQL_C_TINYINT
- SQL_C_SBIGINT
- SQL_C_BIT

Table 192. Converting numeric C data to SQL data

fSQLType	Test	SQLSTATE
SQL_DECIMAL SQL_NUMERIC SQL_SMALLINT SQL_INTEGER SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DOUBLE	Data converted without truncation	N/A
	Data converted with truncation, but without loss of significant digits	22001
	Conversion of data would result in loss of significant digits	22003
SQL_CHAR SQL_VARCHAR	Data converted without truncation.	N/A
	Conversion of data would result in loss of significant digits.	22003

Note: When converting to floating point values, SQLSTATE 22003 will not be returned if non-significant digits of the resulting value are lost.

Converting binary C data to SQL data

The binary C data type is:

- SQL_C_BINARY

Table 193. Converting binary C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Length of data <= Column length	N/A
	Length of data > Column length	22001
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	Length of data <= Column length	N/A
	Length of data > Column length	22001
SQL_XML	Data can be implicitly parsed	(several SQLSTATES can be returned)

Converting DBCHAR C data to SQL data

The double byte C data type is:

- SQL_C_DBCHAR

Table 194. Converting DBCHAR C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Length of data <= Column length x 2	N/A
	Length of data > Column length x 2	22001
SQL_DECIMAL SQL_NUMERIC SQL_SMALLINT SQL_INTEGER SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DECFLOAT SQL_DOUBLE	Length of data <= Column length x 2	N/A
	Length of data > Column length x 2	22001
	Data is non-Unicode	07006
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	Length of data <= Column length x 2	N/A
	Length of data > Column length x 2	22001

Table 194. Converting DBCHAR C data to SQL data (continued)

fSQLType	Test	SQLSTATE
SQL_XML	Data can be implicitly parsed	(several SQLSTATES can be returned)

Converting date C data to SQL data

The date C data type is:

- SQL_C_DATE

Table 195. Converting date C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= 10	N/A
	Column length < 10	22001
SQL_TYPE_DATE	Data value is a valid date	N/A
	Data value is not a valid date	22007
SQL_TYPE_TIMESTAMP[a]	Data value is a valid date	N/A
	Data value is not a valid date	22007

Note: SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

Note: **a**, the time component of TIMESTAMP is set to zero.

Converting time C data to SQL data

The time C data type is:

- SQL_C_TIME

Table 196. Converting time C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= 8	N/A
	Column length < 8	22001
SQL_TYPE_TIME	Data value is a valid time	N/A
	Data value is not a valid time	22007
SQL_TYPE_TIMESTAMP [a]	Data value is a valid time	N/A
	Data value is not a valid time	22007

Note: SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

Note: **a** The date component of TIMESTAMP is set to the system date of the machine at which the application is running.

Converting timestamp C data to SQL data

The timestamp C data type is:

- SQL_C_TIMESTAMP

Table 197. Converting timestamp C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= Display size	N/A
	26 <= Column length < Display size [a]	N/A
	Column length < 26	22001
SQL_TYPE_DATE	Time fields are zero	N/A
	Time fields are non-zero	22008
	Data value does not contain a valid date [b]	22007
	Time fields are non-zero and the connection attribute SQL_ATTR_REPORT_TIMESTAMP_TRUNC_AS_WARN is set to 1	01S07 (Fractional truncation warning)
SQL_TYPE_TIME	Fractional seconds fields are zero	N/A
	Fractional seconds fields are non-zero	22008
	Data value does not contain a valid time	22007
	Time fields are non-zero and the connection attribute SQL_ATTR_REPORT_TIMESTAMP_TRUNC_AS_WARN is set to 1	01S07 (Fractional truncation warning)
SQL_TYPE_TIMESTAMP	Data value is a valid timestamp	N/A
	Data value is not a valid timestamp	22007

Table 197. Converting timestamp C data to SQL data (continued)

fSQLType	Test	SQLSTATE

Note:

[a] The fractional seconds of the timestamp are truncated.

[b] The timestamp_struct must reset the hour, minute, second, and fraction to 0, otherwise SQLSTATE 22008 will be returned.

SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting variable timestamp C data to SQL data

The timestamp C data type is:

- SQL_C_TIMESTAMP_EXT

Table 198. Converting variable timestamp C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= Display size	N/A
	26 <= Column length < Display size [a]	N/A
	Column length < 26	22001
	Fractional seconds fields > 12	22007
SQL_TYPE_DATE	Time fields are zero	N/A
	Time fields are non-zero	22008
	Data value does not contain a valid date [b]	22007
SQL_TYPE_TIME	Fractional seconds fields are zero	N/A
	Fractional seconds fields are non-zero	22008
	Data value does not contain a valid time	22007
SQL_TYPE_TIMESTAMP	Data value is a valid timestamp	N/A
	Data value is not a valid timestamp	22007
	Precision specified by TIMESTAMP(p) <= fractional seconds fields <= 12 [a]	N/A

Note:

[a] The fractional seconds of the timestamp are truncated.

[b] The timestamp_struct must reset the hour, minute, second, and fraction to 0, otherwise SQLSTATE 22008 will be returned.

SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting timestamp(p) with timezone C data to SQL data

The timestamp with timezone C data type is:

- SQL_C_TIMESTAMP_EXT_TZ

Table 199. Converting timestamp with timezone C data to SQL data

fSQLType	Test	SQLSTATE
SQL_CHAR /SQL_VARCHAR SQL_WCHAR/SQL_WVARCHAR SQL_LONGVARCHAR	Column length >= Display size	N/A
	Column length < 27+p	22001
SQL_TYPE_DATE	Time fields are zero	N/A
	Time fields are non-zero	22008
	Data value does not contain a valid date	22007
	Time zone fields are non-zero	22008
SQL_TYPE_TIME	Fractional seconds fields are zero	N/A
	Fractional seconds fields are non-zero	22008
	Data value does not contain a valid time	22007
	Time zone fields are non-zero	22008
SQL_TYPE_TIMESTAMP	Data value is a valid timestamp	N/A
	Data value is not a valid timestamp	22007
	Time zone fields are non-zero	22008
SQL_TYPE_TIMESTAMP_WITH_TIMEZONE	Data value is a valid timestamp	N/A
	Data value is a valid timestamp with time zone	22007

C to SQL data conversion examples

Table 200. C to SQL data conversion examples

C data type	C data value	SQL data type	Column length	SQL data value	SQL STATE
SQL_C_CHAR	abcdef\0	SQL_CHAR	6	abcdef	N/A
SQL_C_CHAR	abcdef\0	SQL_CHAR	5	abcde	22001

Table 200. C to SQL data conversion examples (continued)

C data type	C data value	SQL data type	Column length	SQL data value	SQL STATE
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	6	1234.56	N/A
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	5	1234.5	22001
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	3	---	22003
SQL_C_CHAR	4.46.32	SQL_TYPE_TIME	6	4.46.32	N/A
SQL_C_CHAR	4-46-32	SQL_TYPE_TIME	6	not applicable	22007
SQL_C_DOUBLE	123.45	SQL_CHAR	22	1.23450000 000000e+02	N/A
SQL_C_FLOAT	1234.56	SQL_FLOAT	not applicable	1234.56	N/A
SQL_C_FLOAT	1234.56	SQL_INTEGER	not applicable	1234	22001
SQL_C_ TIMESTAMP	1992-12-31 23:45:55. 123456	SQL_TYPE_DATE	6	1992-12-31	01004
SQL_C_ TIMESTAMP_EXT	2009-06-06 23:45:55. 123456789876	SQL_TYPE_DATE	6	2009-06-06	01004

Note: SQLSTATE **00**000 is not returned by `SQLGetDiagRec()`, rather it is indicated when the function returns SQL_SUCCESS.

Data type attributes

Data type precision (CLI) table

The precision of a numeric column or parameter refers to the maximum number of digits used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum or the defined number of characters of the column or parameter. The following table defines the precision for each SQL data type.

Table 201. Precision

fSqlType	Precision
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length of the column or parameter. For example, the precision of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column or parameter. [a]
SQL_DECIMAL SQL_DECFLOAT SQL_NUMERIC	The defined maximum number of digits. For example, the precision of a column defined as NUMERIC(10,3) is 10 and the precision of a column defined as DECFLOAT(34) is 34.
SQL_SMALLINT [b]	5
SQL_BIGINT	19
SQL_INTEGER [b]	10
SQL_FLOAT [b]	15
SQL_REAL [b]	7
SQL_DOUBLE [b]	15

Table 201. Precision (continued)

fSqlType	Precision
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length of the column or parameter. For example, the precision of a column defined as CHAR(10) FOR BIT DATA, is 10.
SQL_LONGVARBINARY	The maximum length of the column or parameter.
SQL_TYPE_DATE [b]	10 (the number of characters in the yyyy-mm-dd format).
SQL_TYPE_TIME [b]	8 (the number of characters in the hh:mm:ss format).
SQL_TYPE_TIMESTAMP	The number of characters in the "yyyy-mm-dd hh:mm:ss[.fffffffffff]" format used by the TIMESTAMP data type. For example, if a timestamp does not use seconds or fractional seconds, the precision is 16 (the number of characters in the "yyyy-mm-dd hh:mm" format). If a timestamp uses thousandths of a second, the precision is 23 (the number of characters in the "yyyy-mm-dd hh:mm:ss.fff" format).
SQL_TYPE_TIMESTAMP_ WITH_TIMEZONE	The number of characters in the "yyyy-mm-dd hh:mm:ss[.fffffffffff]" format used by the TIMESTAMP_WITH_TIMEZONE data type. The valid range for the time zone is -12:59~ +14:00. If the timezone_hour is negative, the timezone_minute must be negative or zero. If the timezone_hour is positive, the timezone_minute must be positive or zero. If the timezone_hour is zero, the timezone_minute can have any value in the range -59 through +59.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length of the column or parameter. For example, the precision of a column defined as GRAPHIC(10) is 10.
SQL_LONGVARGRAPHIC	The maximum length of the column or parameter.
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	The defined length of the column or parameter. For example, the precision of a column defined as WCHAR(10) is 10.
SQL_XML	0, unless the XML value is an argument to external routines. For external routines, the precision is the defined length, n, of an XML AS CLOB(n) argument.

Note:

[a] When defining the precision of a parameter of this data type with `SQLBindParameter()` or `SQLSetParam()`, *cbParamDef* should be set to the total length of the data, not the precision as defined in this table.

[b] The *cbColDef* argument of `SQLBindParameter()` is ignored for this data type.

Data type scale (CLI) table

The scale of a numeric column or parameter refers to the maximum number of digits to the right of the decimal point. Note that, for approximate floating point number columns or parameters, the scale is undefined, since the number of digits to the right of the decimal place is not fixed. The following table defines the scale for each SQL data type.

Data type scale (CLI) table

Table 202. Scale

fSqlType	Scale
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_CLOB	Not applicable.
SQL_DECIMAL SQL_NUMERIC	The defined number of digits to the right of the decimal place. For example, the scale of a column defined as NUMERIC(10, 3) is 3.
SQL_SMALLINT SQL_INTEGER SQL_BIGINT	0
SQL_REAL SQL_FLOAT SQL_DECFLOAT SQL_DOUBLE	Not applicable.
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB	Not applicable.
SQL_TYPE_DATE SQL_TYPE_TIME	Not applicable.
SQL_TYPE_TIMESTAMP	The number of digits to the right of the decimal point in the "yyyy-mm-dd hh:mm:ss[.fffffffffff]" format. For example, if the TIMESTAMP data type uses the "yyyy-mm-dd hh:mm:ss.fff" format, the scale is 3.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC SQL_DBCLOB	Not applicable.
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Not applicable.
SQL_XML	Not applicable.

Data type length (CLI) table

The length of a column is the maximum number of *bytes* returned to the application when data is transferred to its default C data type. For character data, the length does not include the null termination byte. Note that the length of a column might be different than the number of bytes required to store the data on the data source.

The following table defines the length for each SQL data type.

Table 203. Length

fSqlType	Length
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length of the column. For example, the length of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column.
SQL_DECIMAL SQL_NUMERIC	The maximum number of digits plus two. Since these data types are returned as character strings, characters are needed for the digits, a sign, and a decimal point. For example, the length of a column defined as NUMERIC(10,3) is 12.
SQL_DECFLOAT	If the column is defined as DECFLOAT(16) then the length is 8. If the column is defined as DECFLOAT(34) then the length is 16.
SQL_SMALLINT	2 (two bytes).
SQL_INTEGER	4 (four bytes).
SQL_BIGINT	8 (eight bytes).
SQL_REAL	4 (four bytes).
SQL_FLOAT	8 (eight bytes).
SQL_DOUBLE	8 (eight bytes).
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined length of the column. For example, the length of a column defined as CHAR(10) FOR BIT DATA is 10.
SQL_LONGVARBINARY	The maximum length of the column.
SQL_TYPE_DATE SQL_TYPE_TIME	6 (the size of the DATE_STRUCT or TIME_STRUCT structure).
SQL_TYPE_TIMESTAMP	16 (the size of the TIMESTAMP_STRUCT structure).
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	The defined length of the column times 2. For example, the length of a column defined as GRAPHIC(10) is 20.
SQL_LONGVARGRAPHIC	The maximum length of the column times 2.
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	The defined length of the column times 2. For example, the length of a column defined as WCHAR(10) is 20.

Table 203. Length (continued)

fSqlType	Length
SQL_XML	0 (stored XML documents are limited to 2GB in size however)

Data type display (CLI) table

The display size of a column is the maximum number of *bytes* needed to display data in character form. The following table defines the display size for each SQL data type.

Table 204. Display size

fSqlType	Display size
SQL_CHAR SQL_VARCHAR SQL_CLOB	The defined length of the column. For example, the display size of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column.
SQL_DECIMAL SQL_NUMERIC	The precision of the column plus two (a sign, precision digits, and a decimal point). For example, the display size of a column defined as NUMERIC(10,3) is 12.
SQL_DECFLOAT	If the column is defined as DECFLOAT(16) then the display length is 24. If the column is defined as DECFLOAT(34) then the display length is 42.
SQL_SMALLINT	6 (a sign and 5 digits).
SQL_INTEGER	11 (a sign and 10 digits).
SQL_BIGINT	20 (a sign and 19 digits).
SQL_REAL	13 (a sign, 7 digits, a decimal point, the letter E, a sign, and 2 digits).
SQL_FLOAT SQL_DOUBLE	22 (a sign, 15 digits, a decimal point, the letter E, a sign, and 3 digits).
SQL_BINARY SQL_VARBINARY SQL_BLOB	The defined maximum length of the column times 2 (each binary byte is represented by a 2 digit hexadecimal number). For example, the display size of a column defined as CHAR(10) FOR BIT DATA is 20.
SQL_LONGVARBINARY	The maximum length of the column times 2.
SQL_TYPE_DATE	10 (a date in the format yyyy-mm-dd).
SQL_TYPE_TIME	8 (a time in the format hh:mm:ss).

Table 204. Display size (continued)

fSqlType	Display size
SQL_TYPE_TIMESTAMP	19 (if the scale of the timestamp is 0) or 20 plus the scale of the timestamp (if the scale is greater than 0). This is the number of characters in the "yyyy-mm-dd hh:mm:ss[.fffffffffff]" format. For example, the display size of a column storing thousandths of a second is 23 (the number of characters in "yyyy-mm-dd hh:mm:ss.fff").
SQL_GRAPHIC SQL_VARGRAPHIC SQL_DBCLOB	Twice the defined length of the column or parameter. For example, the display size of a column defined as GRAPHIC(10) is 20.
SQL_LONGVARGRAPHIC	The maximum length of the column or parameter.
SQL_XML	0

Data type display (CLI) table

Appendix A. Overview of the DB2 technical information

DB2 technical information is available in multiple formats that can be accessed in multiple ways.

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics (Task, concept and reference topics)
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF DVD)
  - printed books
- Command-line help
  - Command help
  - Message help

Note: The DB2 Information Center topics are updated more frequently than either the PDF or the hardcopy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks® publications online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how to improve the DB2 documentation, send an email to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this email address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss. English and translated DB2 Version 10.1 manuals in PDF format can be downloaded from www.ibm.com/support/docview.wss?rs=71&uid=swg2700947.

Although the tables identify books available in print, the books might not be available in your country or region.

The form number increases each time a manual is updated. Ensure that you are reading the most recent version of the manuals, as listed below.

Note: The *DB2 Information Center* is updated more frequently than either the PDF or the hard-copy books.

Table 205. DB2 technical information

Name	Form Number	Available in print	Last updated
Administrative API Reference	SC27-3864-00	Yes	April, 2012
Administrative Routines and Views	SC27-3865-00	No	April, 2012
Call Level Interface Guide and Reference Volume 1	SC27-3866-00	Yes	April, 2012
Call Level Interface Guide and Reference Volume 2	SC27-3867-00	Yes	April, 2012
Command Reference	SC27-3868-00	Yes	April, 2012
Database Administration Concepts and Configuration Reference	SC27-3871-00	Yes	April, 2012
Data Movement Utilities Guide and Reference	SC27-3869-00	Yes	April, 2012
Database Monitoring Guide and Reference	SC27-3887-00	Yes	April, 2012
Data Recovery and High Availability Guide and Reference	SC27-3870-00	Yes	April, 2012
Database Security Guide	SC27-3872-00	Yes	April, 2012
DB2 Workload Management Guide and Reference	SC27-3891-00	Yes	April, 2012
Developing ADO.NET and OLE DB Applications	SC27-3873-00	Yes	April, 2012
Developing Embedded SQL Applications	SC27-3874-00	Yes	April, 2012
Developing Java Applications	SC27-3875-00	Yes	April, 2012
Developing Perl, PHP, Python, and Ruby on Rails Applications	SC27-3876-00	No	April, 2012
Developing User-defined Routines (SQL and External)	SC27-3877-00	Yes	April, 2012
Getting Started with Database Application Development	GI13-2046-00	Yes	April, 2012

Table 205. DB2 technical information (continued)

Name	Form Number	Available in print	Last updated
Getting Started with DB2 Installation and Administration on Linux and Windows	GI13-2047-00	Yes	April, 2012
Globalization Guide	SC27-3878-00	Yes	April, 2012
Installing DB2 Servers	GC27-3884-00	Yes	April, 2012
Installing IBM Data Server Clients	GC27-3883-00	No	April, 2012
Message Reference Volume 1	SC27-3879-00	No	April, 2012
Message Reference Volume 2	SC27-3880-00	No	April, 2012
Net Search Extender Administration and User's Guide	SC27-3895-00	No	April, 2012
Partitioning and Clustering Guide	SC27-3882-00	Yes	April, 2012
pureXML Guide	SC27-3892-00	Yes	April, 2012
Spatial Extender User's Guide and Reference	SC27-3894-00	No	April, 2012
SQL Procedural Languages: Application Enablement and Support	SC27-3896-00	Yes	April, 2012
SQL Reference Volume 1	SC27-3885-00	Yes	April, 2012
SQL Reference Volume 2	SC27-3886-00	Yes	April, 2012
Text Search Guide	SC27-3888-00	Yes	April, 2012
Troubleshooting and Tuning Database Performance	SC27-3889-00	Yes	April, 2012
Upgrading to DB2 Version 10.1	SC27-3881-00	Yes	April, 2012
What's New for DB2 Version 10.1	SC27-3890-00	Yes	April, 2012
XQuery Reference	SC27-3893-00	No	April, 2012

Table 206. DB2 Connect-specific technical information

Name	Form Number	Available in print	Last updated
DB2 Connect Installing and Configuring DB2 Connect Personal Edition	SC27-3861-00	Yes	April, 2012
DB2 Connect Installing and Configuring DB2 Connect Servers	SC27-3862-00	Yes	April, 2012
DB2 Connect User's Guide	SC27-3863-00	Yes	April, 2012

Displaying SQL state help from the command line processor

DB2 products return an SQLSTATE value for conditions that can be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

Procedure

To start SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.
For example, `? 08003` displays help for the 08003 SQL state, and `? 08` displays help for the 08 class code.

Accessing different versions of the DB2 Information Center

Documentation for other versions of DB2 products is found in separate information centers on ibm.com®.

About this task

For DB2 Version 10.1 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v10r1.

For DB2 Version 9.8 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r8/.

For DB2 Version 9.7 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/.

For DB2 Version 9.5 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r5.

For DB2 Version 9.1 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9/.

For DB2 Version 8 topics, go to the *DB2 Information Center* URL at: http://publib.boulder.ibm.com/infocenter/db2luw/v8/.

Updating the DB2 Information Center installed on your computer or intranet server

A locally installed DB2 Information Center must be updated periodically.

Before you begin

A DB2 Version 10.1 Information Center must already be installed. For details, see the “Installing the DB2 Information Center using the DB2 Setup wizard” topic in *Installing DB2 Servers*. All prerequisites and restrictions that applied to installing the Information Center also apply to updating the Information Center.

Updating the DB2 Information Center installed on your computer or intranet server

About this task

An existing DB2 Information Center can be updated automatically or manually:

- Automatic updates update existing Information Center features and languages. One benefit of automatic updates is that the Information Center is unavailable for a shorter time compared to during a manual update. In addition, automatic updates can be set to run as part of other batch jobs that run periodically.
- Manual updates can be used to update existing Information Center features and languages. Automatic updates reduce the downtime during the update process, however you must use the manual process when you want to add features or languages. For example, a local Information Center was originally installed with both English and French languages, and now you want to also install the German language; a manual update will install German, as well as, update the existing Information Center features and languages. However, a manual update requires you to manually stop, update, and restart the Information Center. The Information Center is unavailable during the entire update process. In the automatic update process the Information Center incurs an outage to restart the Information Center after the update only.

This topic details the process for automatic updates. For manual update instructions, see the "Manually updating the DB2 Information Center installed on your computer or intranet server" topic.

Procedure

To automatically update the DB2 Information Center installed on your computer or intranet server:

1. On Linux operating systems,
   a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `/opt/ibm/db2ic/V10.1` directory.
   b. Navigate from the installation directory to the `doc/bin` directory.
   c. Run the `update-ic` script:
      ```
      update-ic
      ```
2. On Windows operating systems,
   a. Open a command window.
   b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `<Program Files>\IBM\DB2 Information Center\Version 10.1` directory, where `<Program Files>` represents the location of the Program Files directory.
   c. Navigate from the installation directory to the `doc\bin` directory.
   d. Run the `update-ic.bat` file:
      ```
      update-ic.bat
      ```

Results

The DB2 Information Center restarts automatically. If updates were available, the Information Center displays the new and updated topics. If Information Center updates were not available, a message is added to the log. The log file is located in `doc\eclipse\configuration` directory. The log file name is a randomly generated number. For example, `1239053440785.log`.

Manually updating the DB2 Information Center installed on your computer or intranet server

If you have installed the DB2 Information Center locally, you can obtain and install documentation updates from IBM.

About this task

Updating your locally installed *DB2 Information Center* manually requires that you:

1. Stop the *DB2 Information Center* on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to apply updates. The Workstation version of the DB2 Information Center always runs in stand-alone mode. .
2. Use the Update feature to see what updates are available. If there are updates that you must install, you can use the Update feature to obtain and install them

   Note: If your environment requires installing the *DB2 Information Center* updates on a machine that is not connected to the internet, mirror the update site to a local file system by using a machine that is connected to the internet and has the *DB2 Information Center* installed. If many users on your network will be installing the documentation updates, you can reduce the time required for individuals to perform the updates by also mirroring the update site locally and creating a proxy for the update site.
   If update packages are available, use the Update feature to get the packages. However, the Update feature is only available in stand-alone mode.
3. Stop the stand-alone Information Center, and restart the *DB2 Information Center* on your computer.

Note: On Windows 2008, Windows Vista (and higher), the commands listed later in this section must be run as an administrator. To open a command prompt or graphical tool with full administrator privileges, right-click the shortcut and then select **Run as administrator**.

Procedure

To update the *DB2 Information Center* installed on your computer or intranet server:

1. Stop the *DB2 Information Center*.
   - On Windows, click **Start** > **Control Panel** > **Administrative Tools** > **Services**. Then right-click **DB2 Information Center** service and select **Stop**.
   - On Linux, enter the following command:
     ```
     /etc/init.d/db2icdv10 stop
     ```
2. Start the Information Center in stand-alone mode.
   - On Windows:
     a. Open a command window.
     b. Navigate to the path where the Information Center is installed. By default, the *DB2 Information Center* is installed in the *Program_Files*\IBM\DB2 Information Center\Version 10.1 directory, where *Program_Files* represents the location of the Program Files directory.
     c. Navigate from the installation directory to the doc\bin directory.
     d. Run the help_start.bat file:

```
help_start.bat
```

- On Linux:
  a. Navigate to the path where the Information Center is installed. By default, the *DB2 Information Center* is installed in the `/opt/ibm/db2ic/V10.1` directory.
  b. Navigate from the installation directory to the `doc/bin` directory.
  c. Run the `help_start` script:

     ```
     help_start
     ```

  The systems default Web browser opens to display the stand-alone Information Center.

3. Click the **Update** button ( ). (JavaScript must be enabled in your browser.) On the right panel of the Information Center, click **Find Updates**. A list of updates for existing documentation displays.
4. To initiate the installation process, check that the selections you want to install, then click **Install Updates**.
5. After the installation process has completed, click **Finish**.
6. Stop the stand-alone Information Center:
   - On Windows, navigate to the `doc\bin` directory within the installation directory, and run the `help_end.bat` file:

     ```
     help_end.bat
     ```

     Note: The `help_end` batch file contains the commands required to safely stop the processes that were started with the `help_start` batch file. Do not use `Ctrl-C` or any other method to stop `help_start.bat`.
   - On Linux, navigate to the `doc/bin` directory within the installation directory, and run the `help_end` script:

     ```
     help_end
     ```

     Note: The `help_end` script contains the commands required to safely stop the processes that were started with the `help_start` script. Do not use any other method to stop the `help_start` script.
7. Restart the *DB2 Information Center*.
   - On Windows, click **Start** > **Control Panel** > **Administrative Tools** > **Services**. Then right-click **DB2 Information Center** service and select **Start**.
   - On Linux, enter the following command:

     ```
     /etc/init.d/db2icdv10 start
     ```

Results

The updated *DB2 Information Center* displays the new and updated topics.

DB2 tutorials

The DB2 tutorials help you learn about various aspects of DB2 database products. Lessons provide step-by-step instructions.

Before you begin

You can view the XHTML version of the tutorial from the Information Center at http://publib.boulder.ibm.com/infocenter/db2luw/v10r1/.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

DB2 tutorials

To view the tutorial, click the title.

"pureXML® " in *pureXML Guide*
Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 database products.

DB2 documentation
Troubleshooting information can be found in the *Troubleshooting and Tuning Database Performance* or the Database fundamentals section of the *DB2 Information Center*, which contains:

- Information about how to isolate and identify problems with DB2 diagnostic tools and utilities.
- Solutions to some of the most common problem.
- Advice to help solve other problems you might encounter with your DB2 database products.

IBM Support Portal
See the IBM Support Portal if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.

Access the IBM Support Portal at http://www.ibm.com/support/entry/portal/Overview/Software/Information_Management/DB2_for_Linux,_UNIX_and_Windows

Terms and conditions

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability: These terms and conditions are in addition to any terms of use for the IBM website.

Personal use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights: Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Trademarks: IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.shtml

Appendix B. Notices

This information was developed for products and services offered in the U.S.A. Information about non-IBM products is based on information available at the time of first publication of this document and is subject to change.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information about the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements, changes, or both in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to websites not owned by IBM are provided for convenience only and do not in any manner serve as an endorsement of those

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information that has been exchanged, should contact:

IBM Canada Limited
U59/3600
3600 Steeles Avenue East
Markham, Ontario L3R 9Z7
CANADA

Such information may be available, subject to appropriate terms and conditions, including, in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies

- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle, its affiliates, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Intel, Intel logo, Intel Inside, Intel Inside logo, Celeron, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Printed in USA

SC27-3867-00

Spine information:

IBM DB2 10.1 for Linux, UNIX, and Windows

Call Level Interface Guide and Reference Volume 2